图灵原创

从源代码的角度深入剖析Storm设计与实现

微软搜索技术部门高级研发工程师实战经验

学习大牛如何实现和高效利用“实时的Hadoop”

Storm源码分析

李明 王晓鹏 ◎ 编著



人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

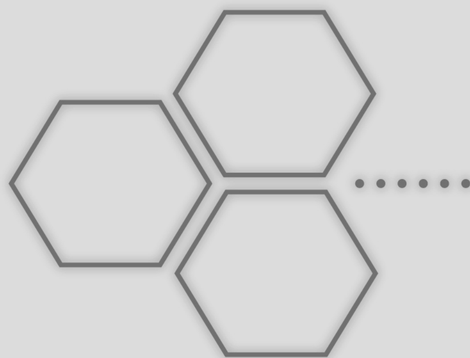
如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

作者简介

李明 2001~2007年在哈尔滨工业大学完成本科以及硕士的学习。微软搜索技术部门资深研发工程师及高级开发经理，擅长Linux、Clojure、Java、C#等多种开发技术，长期致力于大数据、分布式系统的研究和应用，目前致力于实时性分布式处理系统的研究与开发。

王晓鹏 2004~2011年在北京邮电大学完成本科以及硕士的学习。微软搜索技术部门高级研发工程师，擅长Windows Phone、Silverlight、Clojure、Java、C#等多种开发技术，一直致力于大数据处理、分布式系统的研究和应用，目前致力于实时性分布式处理系统的研究与开发。

图灵原创



Storm源码分析

李明　王晓鹏 ◎ 编著

人民邮电出版社

北京

图书在版编目（CIP）数据

Storm源码分析 / 李明，王晓鹏编著. -- 北京 : 人民邮电出版社，2014.10
（图灵原创）
ISBN 978-7-115-37126-3

Ⅰ. ①S… Ⅱ. ①李… ②王… Ⅲ. ①数据处理软件
Ⅳ. ①TP274

中国版本图书馆CIP数据核字(2014)第221005号

内 容 提 要

本书从源代码的角度详细分析了 Storm 的设计与实现，共分为三个部分，第一部分介绍了 Storm 的基本原理以及 Storm 集群系统的搭建方法，第二部分深入剖析了 Storm 的底层架构，如 Nimbus、Supervisor、Worker 以及 Task，第三部分系统讨论了 Storm 如何实现可靠的消息传输，如 Transaction Topology 以及 Trident。

本书适用于程序员、架构师以及计算机专业的学生。

◆ 编　　著　李　明　王晓鹏
责任编辑　王军花
执行编辑　李鸿鹏
责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：800×1000　1/16
印张：30.25
字数：715千字　　　　2014年 10 月第 1 版
印数：1 – 3 000册　　　　2014年 10 月北京第 1 次印刷

定价：79.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

专家推荐

流计算是目前计算机领域非常热门的技术，Storm 平台的出现大大推进了该项技术的发展，并被很多包括微软在内的大公司采用。《Storm 源代码分析》从源代码角度深入浅出地分析了 Storm 的设计及实现，一方面可以使读者更好地了解并用好 Storm 技术，另一方面可以让读者学习如何设计大规模分布式系统，相信读者一定会受益匪浅。

——于伟，微软资深开发总监

在当今互联网众多领域中，大数据和云计算无疑是两个最火的主题，而当中尤其以大数据的实时流处理为很多开发者都感兴趣的。作者在书中对 Storm 进行了详尽的介绍，按部就班，化繁为简，让读者能一步一景地学懂 Storm 的箇中细节，实在是 Storm 入门者的必备良药。

在我十多载微软职业生涯当中（美国总部和中国），遇到的技术大牛多如繁星，但李明和晓鹏给我的印象尤为深刻。记得当初我们组建广告 BI 团队的时候就先立下了采用开源技术这个指导思想，对传统的微软人来说，这是开创先河之举。李明和晓鹏是最早加入的，从第一天开始，我就感觉到他们对技术的热情与执着。在短短的一个月里，他们不仅理解了 Storm 的精髓和关键，还实现了一个 BI pipeline 的雏形，让我们能展现出实时流处理大数据的力量。在往后的日子里，每天的工作量都非常庞大，但更加令我惊讶的是，他们居然写了一本关于 Storm 的书。要知道学懂一门技术对于开发者来说不难，但要著书立说却要经过一定程度的沉淀和思考，并非旦夕之功。我拜读了他们的初稿后，发觉书中对 Storm 的理解精辟透彻，对 Storm 的运用和各处细节也都阐述入微。尤其是对 Storm 的入门初学者来说，是一本不可多得的好书。在后续的日子里，当我们构建大数据平台时，他们已经转岗到别的团队，但我们在遇到技术难点的时候，还会借助他们对 Storm 的深刻了解来解决问题。这是一本应该放在桌面、随时可以翻阅的参考书。赞！

——章英基，前微软资深开发总监，现阿里巴巴资深总监

大数据处理是当前计算机科技的热点，而流式实时大数据处理更是这皇冠上璀璨的明珠。实时流数据处理在搜索引擎、社交网络、电商网站、广告平台等领域有着相当广泛的应用。Storm 是极其高效、灵活、高扩展的流式数据处理平台，它被 Twitter、Taobao、Yahoo！和 Groupon 等公司采用。

本书由微软公司互联网工程院经验丰富的一线程序员操刀编写，包含很多实战经验和使用心得，很好地结合了代码分析和应用实例。本书对于进行流式数据处理的研究、Storm 的深入理解

以及实际应用都有很好的参考价值。我有幸与本书作者李明、王晓鹏共同开发基于 Storm 的应用，他们深厚的程序功力、不懈的钻研精神令我深深地叹服，这在本书中也得到了很好的体现。我相信读者一定会受益匪浅。

——王明雨，微软资深开发工程师

本书是国内为数不多的讲述大数据流计算并进行源码分析的一本实战书。它出自技术精湛并且具有丰富实战经验的的微软工程师之手，我相信绝不会让你失望！作为解决大数据 5 个"V"之一的"Velocity"问题，Storm 是最流行的实时计算框架，它被认为是 Hadoop 批处理计算的补充，它比 Map/Reduce 更加灵活，而且性能、可靠性和可扩展性出众，所以在 Twitter 等互联网公司被广泛应用到大数据实时和准实时处理的生产环境。

在微软公司担任数据平台产品经理期间，我有幸和李明、王晓鹏等同事合作，一起将 Storm 应用于微软搜索中心的广告、监控、安全和预测等应用场景。在工作期间，这本书对我帮助很大，即便对于像我这样在分布式领域工作 12 年的老手来讲，这本书仍然让我受益良多。无论你是大数据领域、分布式系统的从业人员，还是开源系统的爱好者、开发者或互联网从业人员，我认为这本书都值得仔细研读。如果你想了解流计算的设计原理，想洞悉 Storm 的设计精髓，或揣摩用 Clojure，抑或是探求如何用 Storm 来解决大数据的准实时需求场景，这本书都会对你大有裨益。

——贺军，微软资深项目经理

前　言

Storm是一个分布式的、可靠的实时计算系统。与Hadoop的批处理不同，Storm采用流式的消息处理方法，它使得消息可以得到快速的处理，可以用于实时性要求较高的系统，例如广告点击的在线统计等。Storm弥补了Hadoop在实时处理方面的缺陷，目前被各大互联网公司广泛使用并日益流行。

本书作为第一本深入介绍Storm的图书，从源代码的角度详细剖析了Storm的设计与实现。本书适合各类型的计算机工作者，初学者可以通过本书来学习如何实现一个可靠的、高容错性的、实时的分布式处理平台。而对于Storm用户来讲，本书不仅可以帮助他们更深入地了解这套系统的工作原理，还可以帮助他们正确地使用该平台，也有利于实现对Storm的二次开发。鉴于Storm是基于Clojure和Java开发的，所以需要读者对这两种语言有一定的了解。

本书主要分析阐述了Storm的底层架构，例如Nimbus、Supervisor、Worker、Executor以及Task，并对Storm如何实现可靠的消息传输进行了系统讨论，例如事务Topology以及Trident。

本书对Storm的最新源代码进行了系统而详尽的分析，相信读者在阅读过程中一定会获益匪浅。

致谢

诚挚感谢人民邮电出版社和图灵公司为我提供创作的平台。感谢本书的编辑王军花、张霞等，你们的专业态度和细致工作提升了本书的品质。

诚挚感谢妻子包黎云，没有她的支持，我无法完成本书。最后，将该书献给我即将出世的孩子。

——李明

诚挚感谢同事贺军、王明雨以及童杰，感谢你们在这本书的编写过程中给予我们的无私帮助。特别感谢我的妻子白鸽，感谢你对我的理解、包容和支持！

——晓鹏

目　录

第1章 总体架构与代码结构

Storm是由BackType开发的一个实时、分布式的计算平台，后来Twitter收购了BackType并将其源代码开放。在GitHub上（https://github.com/nathanmarz/storm），我们可以获得Storm的最新源代码及相关文档。

1.1 Storm 的总体结构

Storm中会涉及的术语包括Stream、Spout、Bolt、Worker、Executor、Task、Stream Grouping和Topology，现简要介绍如下。

- Stream是被处理的数据。
- Spout是数据源。
- Bolt封装了数据处理逻辑。
- Worker是工作进程。一个工作进程中可以含有一个或多个Executor线程。
- Executor是运行Spout或Bolt处理逻辑的线程。
- Task是Storm中的最小处理单元。一个Executor中可以包含一个或多个Task，消息的分发都是从一个Task到另一个Task进行的。
- Stream Grouping定义了消息分发策略，定义了Bolt节点以何种方式接收数据。消息可以随机分配（Shuffle Grouping，随机分组），或者根据字段值分配（Fields Grouping，字段分组），或者广播（All Grouping，全部分组），或者总是发给同一个Task（Global Grouping，全局分组），也可以不关心数据是如何分组的（None Grouping，无分组），或者由自定义逻辑来决定，即由消息发送者决定应该由消息接收者组件的哪个Task来处理该消息（Direct Grouping，直接分组）。
- Topology是由消息分组方式连接起来的Spout和Bolt节点网络，它定义了运算处理的拓扑结构，处理的是不断流动的消息。除非杀掉Topology，否则它将永远运行下去。

Storm的基本结构如图1-1所示。

Storm集群中存在两种类型的节点：运行Nimbus服务的主节点和运行Supervisor服务的工作节点。Storm集群由一个主节点和多个工作节点组成。主节点上运行一个名为“Nimbus”的守护进程，用于分配代码、布置任务及检测故障。每个工作节点则运行一个名为“Supervisor”的守护

进程，用于监听工作、开始并终止工作进程。

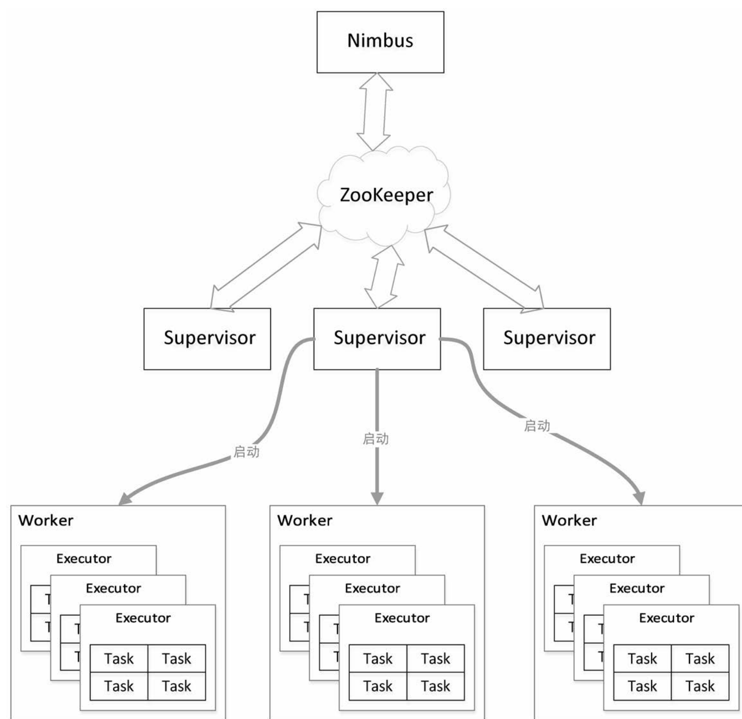


图1-1　Storm的基本结构

Nimbus和Supervisor都能快速失败并恢复，而且它们是无状态的，其元数据存储在ZooKeeper中，这使得系统具有很高的容错性。Nimbus与Supervisor之间的协调工作是通过ZooKeeper来完成的，它是Apache下面的开源项目，用于分布式系统的同步等，详情可参考http://zookeeper.apacheorg/。

Worker由Supervisor负责启动，一个Worker中可以有多个Executor线程，每个Executor中又可包含一个或多个Task。Task为Storm中的最小处理单元，它是Topology组件诸多并行度中的一个。每个Executor都会启动一个消息循环线程，用以接收、处理和发送消息。当Executor收到属于其下某一Task的消息后，就会调用该Task对应的处理逻辑对消息进行处理。

在逻辑上，Storm中消息的来源节点被称为Spout，消息的处理节点被称为Bolt。在系统中，可以存在多个Spout及Bolt，且每个Spout或Bolt都可设置不同的并行度，示例如图1-2所示。

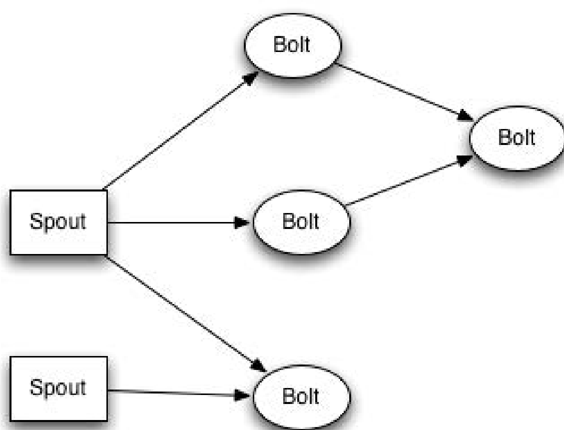


图1-2 示例图

1.2 Storm 的元数据

Storm采用ZooKeeper来存储Nimbus、Supervisor、Worker以及Executor之间共享的元数据，这些模块在重启之后，可以通过对应的元数据进行恢复。因此Storm的模块是无状态的，这是保证其可靠性及可扩展性的基础。了解元数据以及Storm如何使用这些元数据，有助于我们更好地理解Storm的设计。

1.2.1 元数据介绍

Storm在ZooKeeper中存储数据的目录结构如图1-3所示，这是一个根路径为/storm的树，树中的每一个节点代表ZooKeeper中的一个节点（znode），每一个叶子节点是Storm真正存储数据的地方。在图1-3中，从根节点到叶子节点的全路径代表了该数据在ZooKeeper中的存储路径，该路径可被用来写入或获取数据。

下面分别介绍ZooKeeper中每项数据的具体含义。

- /storm/workerbeats/<topology-id>/node-port：它存储由node和port指定的Worker的运行状态和一些统计信息，主要包括storm-id（也即topology-id）、当前Worker上所有Executor的统计信息（如发送的消息数目、接收的消息数目等）、当前Worker的启动时间以及最后一次更新这些信息的时间。在一个topology-id下面，可能有多个node-port节点。它的内容在运行过程中会被更新。
- /storm/storms/<topology-id>：它存储Topology本身的信息，包括它的名字、启动时间、运行状态、要使用的Worker数目以及每个组件的并行度设置。它的内容在运行过程中是不变的。

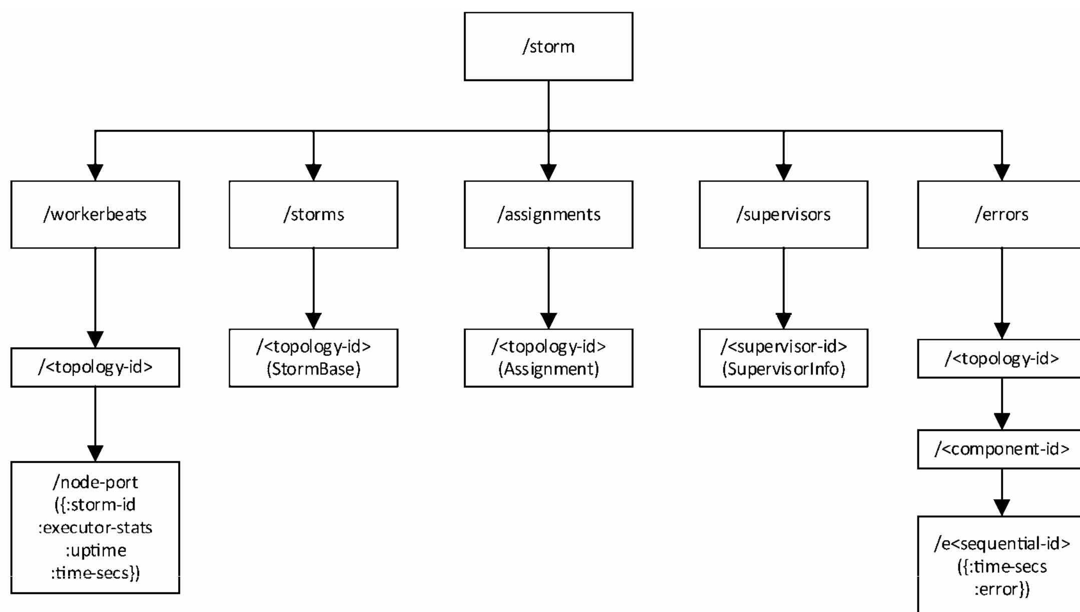


图1-3　Storm在ZooKeeper中存储的数据

- /storm/assignments/<topology-id>：它存储了Nimbus为每个Topology分配的任务信息，包括该Topology在Nimbus机器本地的存储目录、被分配到的Supervisor机器到主机名的映射关系、每个Executor运行在哪个Worker上以及每个Executor的启动时间。该节点的数据在运行过程中会被更新。
- /storm/supervisors/<supervisor-id>：它存储Supervisor机器本身的运行统计信息，主要包括最近一次更新时间、主机名、supervisor-id、已经使用的端口列表、所有的端口列表以及运行时间。该节点的数据在运行过程中也会被更新。
- /storm/errors/<topology-id>/<component-id>/e<sequential-id>：它存储运行过程中每个组件上发生的错误信息。<sequential-id>是一个递增的序列号，每一个组件最多只会保留最近的10条错误信息。它的内容在运行过程中是不变的（但是有可能被删除）。

1.2.2　Storm怎么使用这些元数据

了解了存储在ZooKeeper中的数据，我们自然想知道Storm是如何使用这些元数据的。例如，这些数据何时被写入、更新或删除，这些数据都是由哪种类型的节点（Nimbus、Supervisor、Worker或者Executor）来维护的。接下来，我们就简单介绍一下这些关系，希望读者能对Storm的整体设计实现有更深一层的认识。带上这些知识，能让你的Storm源码之路变得更加轻松愉快。

首先来看一下总体交互图，如图1-4所示。

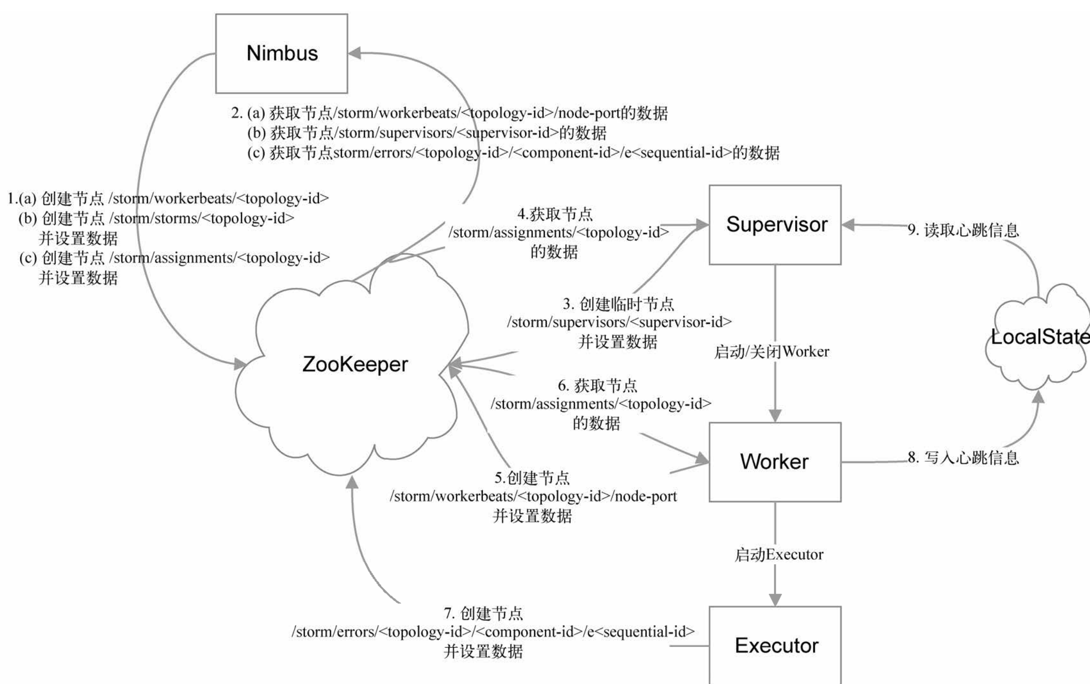


图1-4 总体交互图

这个图描述了Storm中每个节点跟ZooKeeper内元数据之间的读写依赖关系，详细介绍如下。

1. Nimbus

Nimbus既需要在ZooKeeper中创建元数据，也需要从ZooKeeper中获取元数据。下面简述图1-4中箭头1和箭头2的作用。

- 箭头1表示由Nimbus创建的路径，包括：

  a. /storm/workerbeats/<topology-id>

  b. /storm/storms/<topology-id>

  c. /storm/assignments/<topology-id>

其中对于路径a，Nimbus只会创建路径，不会设置数据（数据是由Worker设置的，后面会介绍）；对于路径b和c，Nimbus在创建它们的时候就会设置数据。a和b只有在提交新Topology的时候才会创建，且b中的数据设置好后就不再变化，c则在第一次为该Topology进行任务分配的时候创建，若任务分配计划有变，Nimbus就会更新它的内容。

- 箭头2表示Nimbus需要获取数据的路径，包括：

  a. /storm/workerbeats/<topology-id>/node-port

  b. /storm/supervisors/<supervisor-id>

  c. /storm/errors/<topology-id>/<component-id>/e<sequential-id>

Nimbus需要从路径a读取当前已被分配的Worker的运行状态。根据该信息，Nimbus可以得知

哪些Worker状态正常，哪些需要被重新调度，同时还会获取到该Worker所有Executor统计信息，这些信息会通过UI呈现给用户。从路径b可以获取当前集群中所有Supervisor的状态，通过这些信息可以得知哪些Supervisor上还有空闲的资源可用，哪些Supervisor则已经不再活跃，需要将分配到它的任务分配到其他节点上。从路径c上可以获取当前所有的错误信息并通过UI呈现给用户。集群中可以动态增减机器，机器的增减会引起ZooKeeper中元数据的变化，Nimbus通过不断获取这些元数据信息来调整任务分配，故Storm具有良好的可扩展性。当Nimbus死掉时，其他节点是可以继续工作的，但是不能提交新的Topology，也不能重新进行任务分配和负载调整，因此目前Nimbus还是存在单点的问题。

2. Supervisor

同Nimbus类似，Superviser也要通过ZooKeeper来创建和获取元数据。除此之外，Supervisor还通过监控指定的本地文件来检测由它启动的所有Worker的运行状态。下面简述图1-4中箭头3、箭头4和箭头9的作用。

- 箭头3表示Supervisor在ZooKeeper中创建的路径是/storm/supervisors/<supervisor-id>。新节点加入时，会在该路径下创建一个节点。值得注意的是，该节点是一个临时节点（创建ZooKeeper节点的一种模式），即只要Supervisor与ZooKeeper的连接稳定存在，该节点就一直存在；一旦连接断开，该节点则会被自动删除。该目录下的节点列表代表了目前活跃的机器。这保证了Nimbus能及时得知当前集群中机器的状态，这是Nimbus可以进行任务分配的基础，也是Storm具有容错性以及可扩展性的基础。
- 箭头4表示Supervisor需要获取数据的路径是/storm/assignments/<topology-id>。我们知道它是Nimbus写入的对Topology的任务分配信息，Supervisor从该路径可以获取到Nimbus分配给它的所有任务。Supervisor在本地保存上次的分配信息，对比这两部分信息可以得知分配信息是否有变化。若发生变化，则需要关闭被移除任务所对应的Worker，并启动新的Worker执行新分配的任务。Nimbus会尽量保持任务分配的稳定性，我们将在第7章中进行详细分析。
- 箭头9表示Supervisor会从`LocalState`（相关内容会在第4章中介绍）中获取由它启动的所有Worker的心跳信息。Supervisor会每隔一段时间检查一次这些心跳信息，如果发现某个Worker在这段时间内没有更新心跳信息，表明该Worker当前的运行状态出了问题。这时Supervisor就会杀掉这个Worker，原本分配给这个Worker的任务也会被Nimbus重新分配。

3. Worker

Worker也需要利用ZooKeeper来创建和获取元数据，同时它还需要利用本地的文件来记录自己的心跳信息。

下面简述图4-1中箭头5、箭头6和箭头8的作用。

- 箭头5表示Worker在ZooKeeper中创建的路径是/storm/workerbeats/<topology-id>/node-port。在Worker启动时，将创建一个与其对应的节点，相当于对自身进行注册。需要注意的是，Nimbus在Topology被提交时只会创建路径/storm/workerbeats/<topology-id>，而不会设置数据，数据则留到Worker启动之后由Worker创建。这样安排的目的之一是为了避免

多个Worker同时创建路径时所导致的冲突。

- 箭头6表示Worker需要获取数据的路径是/storm/assignments/<topology-id>，Worker会从这些任务分配信息中取出分配给它的任务并执行。
- 箭头8表示Worker在LocalState中保存心跳信息。LocalState实际上将这些信息保存在本地文件中，Worker用这些信息跟Supervisor保持心跳，每隔几秒钟需要更新一次心跳信息。Worker与Supervisor属于不同的进程，因而Storm采用本地文件的方式来传递心跳。

4. Executor

Executor只会利用ZooKeeper来记录自己的运行错误信息，下面简述图4-1中箭头7的作用。

箭头7表示Executor在ZooKeeper中创建的路径是/storm/errors/<topology-id>/<component-id>/e<sequential-id>。每个Executor会在运行过程中记录发生的错误。

5. 小结

从前面的描述中可以得知，Nimbus、Supervisor以及Worker两两之间都需要维持心跳信息，它们的心跳关系如下。

- Nimbus和Supervisor之间通过/storm/supervisors/<supervisor-id>路径对应的数据进行心跳保持。Supervisor创建这个路径时采用的是临时节点模式，所以只要Supervisor死掉，对应路径的数据就会被删掉，Nimbus就会将原本分配给该Supervisor的任务重新分配。
- Worker跟Nimbus之间通过/storm/workerbeats/<topology-id>/node-port中的数据进行心跳保持。Nimbus会每隔一定时间获取该路径下的数据，同时Nimbus还会在它的内存中保存上一次的信息。如果发现某个Worker的心跳信息有一段时间没更新，就认为该Worker已经死掉了，Nimbus会对任务进行重新分配，将分配至该Worker的任务分配给其他Worker。
- Worker跟Supervisor之间通过本地文件（基于LocalState）进行心跳保持。

1.3 Storm 的代码结构

在本书中，我们主要分析Storm 0.9.0的源代码，其下载地址为https://github.com/nathanmarz/storm/tree/0.9.0。

Storm的源代码主要基于Clojure以及Java来完成，下面简要介绍一下主要的名字空间。

1.3.1 Clojure代码

这部分代码为Storm基础架构的实现，其中Nimbus、Supervisor、Worker、Executor以及Task这些基础组件的实现位于src\clj\backtype.storm下，如表1-1所示。

表1-1 Clojure代码

命名空间	描　述
backtype.storm.daemon.acker	AckerBolt的实现
backtype.storm.daemon.builtin-metrics	Storm内置统计信息

（续）

命名空间	描 述
backtype.storm.daemon.common	包含一些工具方法以及一些全局定义
backtype.storm.daemon.drpc	Storm DRPC服务器的实现
backtype.storm.daemon.executor	Storm Executor的实现
backtype.storm.daemon.nimbus	Storm Nimbus服务的实现
backtype.storm.daemon.supervisor	Storm Supervisor的实现
backtype.storm.daemon.task	Storm Task的实现
backtype.storm.daemon.worker	Storm Worker的实现
backtype.storm.messaging	Storm的底层消息传输封装，主要对ZMQ进行封装
backtype.storm.scheduler	Storm的Nimbus使用的任务调度算法实现
backtype.storm.stats	Storm的运行统计
backtype.storm.ui	Storm UI的实现
backtype.storm.disruptor	Storm中Disruptor Queue的使用包装
backtype.storm.cluster	Storm中对ZooKeeper的使用包装
backtype.storm.zookeeper	基于CuratorFramework的ZooKeeper使用工具类
backtype.storm.util	Clojure工具方法集合，该命名空间含有很多的Clojure工具方法
backtype.storm.timer	基于线程的定时器实现
backtype.storm.config	Storm中配置项读取以及ZooKeeper中元数据路径
backtype.storm.command	Topology的操作，如暂停、杀死等

1.3.2 Java代码

这部分代码含有Storm基础的流处理以及事务Topology的实现，位于src\jvm\backtype.storm下，如表1-2所示。

表1-2 Java代码

命名空间	描 述
backtype.storm.coordination	在Storm流处理的基础上实现批处理。DRPC以及事务Topology都需要使用，CoordinatedBolt是其中重要的类实现
backtype.storm.drpc	DRPC的高层抽象实现，例如DRPCSpout
backtype.storm.generated	通过storm.thrift产生的代码，用于实现Nimbus的服务以及基础数据结构定义
backtype.storm.grouping	包含用户自定义的分组方式接口
backtype.storm.hooks	含有系统的钩子方法接口，用户可以定义钩子函数并被Storm在适当时机调用
backtype.storm.metric	含有信息统计Bolt的接口以及SystemBolt实现

（续）

命名空间	描　　述
backtype.storm.nimbus	Topology有效性检查的接口定义，用于Nimbus
backtype.storm.scheduler	Nimbus任务调度算法相关的接口
backtype.storm.serialization	序列化，主要对Kryo的使用进行封装，提供工具类用于序列化以及反序列化传输的消息
backtype.storm.spout	定义Spout相关接口
backtype.storm.task	定义Bolt以及相关接口，定义相关的上下文对象
backtype.storm.testing	用于测试的一些Spout/Bolt定义以及工具方法
backtype.storm.topology	用于构建Storm Topology的Java API
backtype.storm.transactional	事务Topology的实现
backtype.storm.tuple	Storm的消息数据模型
backtype.storm.utils	Java工具方法及数据结构
backtype.storm.Config类	Storm中用到的参数定义以及含义
backtype.storm.Constant类	系统内置的组件以及流定义

1.3.3　Trident代码

Trident是Storm对实时消息处理的更高层抽象，是Storm的发展方向之一，详情可参见https://github.com/nathanmarz/storm/wiki/Trident-tutorial。Trident代码位于src\jvm\storm.trident下，具体如表1-3所示。

表1-3　Trident代码

命名空间	描　　述
storm.trident.drpc	用于向DRPC服务器返回结果的类实现
storm.trident.fluent	DRPC的多聚集器操作及分组流
storm.trident.graph	Topology对应的有向图的构建工具类
storm.trident.operation	定义Trident的操作
storm.trident.partition	Trident的自定义分组算法
storm.trident.planner	Topology的执行优化
storm.trident.spout	Trident中Spout封装定义
storm.trident.state	Trident中的存储
storm.trident.topology	Trident的Topology构建Java API，以及一些用于同步的重要类定义，例如TridentBolt Executor
storm.trident.tuple	Trident的消息封装
storm.trident.stream类	Trident中对流的抽象，是Trident的核心概念
storm.trident.TridentTopology类	用于定义TridentTopology

1.3.4 其他代码

除了以上介绍的三部分之外，Storm还定义了一些基础工具类以及扩展类，主要包括以下几项。

- storm.thrift以及genthrift.sh：Storm基础数据结构和服务的Thrift定义文件以及产生脚本。
- src\ui：定义了Storm UI的资源文件。
- src\multilang：Storm的多语言支持示例。
- src\clj\zilch\mq.clj：ZMQ的使用包装。

访问下面的链接以获得代码库结构的相关信息：

https://github.com/nathanmarz/storm/wiki/Structure-of-the-codebase。

第2章

搭建Storm集群

在开始研究Storm之前，我们先来搭建单机的和多机的Storm运行环境，然后提交一个示例Topology到搭建好的集群上使其运行，最后再分析一下这个示例Topology的组成。通过学习这一章并进行实践，读者不仅可以对Storm有一个初步的认识，而且能进一步了解Storm的运行原理。

2.1 搭建单机 Storm 集群

为了更好地理解Storm的运行原理，我们可以在单机搭建一套Storm运行环境来模拟真实的Storm集群，这有助于我们进一步了解Storm的运行机制，同时还可以基于它进行本地调试。

下面我们会一步一步介绍如何搭建一个本地的Storm运行环境（操作系统版本是Ubuntu 12.04）。

(1) 下载所需的资源

搭建Storm本地运行环境时，需要下载的资源如下所示。

- Storm：http://storm-project.net/downloads.html，版本0.9.0rc2。
- ZooKeeper：http://www.apache.org/dyn/closer.cgi/zookeeper/，版本3.4.5。
- ZMQ：http://download.zeromq.org/，版本2.1.7。
- jzmq：https://github.com/nathanmarz/jzmq/archive/master.zip。

我们将下载完的所有文件保存在~/project/Storm/文件夹下，并分别解压，相关代码如下。

```
tar -xvf zookeeper-3.4.5.tar.gz
tar -xvf zookeeper-3.4.5.tar.gz
unzip jzmq-master.zip
unzip storm-0.9.0-rc2.zip
```

(2) 安装JDK

这里我们使用openjdk，安装命令是：

```
sudo apt-get install openjdk-7-jdk
```

安装完成后，需要为其设置环境变量。修改文件~/.profile，添加以下内容：

```
export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-i386
export CLASSPATH=$JAVA_HOME/lib
export PATH=$JAVA_HOME/bin:$PATH
```

保存修改后，运行如下命令使其立即生效：

```
source ~/.profile
```

(3) 安装依赖的库文件

Storm及其组件需要依赖很多库文件才能正常工作。依次运行以下命令，安装所有的库：

```
sudo apt-get install libtool
sudo apt-get install autoconf
sudo apt-get install automake
sudo apt-get install g++
sudo apt-get install uuid-dev
sudo apt-get install uuid
sudo apt-get install e2fsprogs
sudo apt-get install python
```

(4) 安装ZMQ

进入解压后的zeromq-2.1.7文件夹，依次运行以下命令：

```
./configure
make
sudo make install
sudo ldconfig
```

(5) 安装jzmq

由于ZMQ是C/C++的库文件，Storm是基于JVM的，没办法直接使。jzmq是用JNI封装的ZMQ的Java库，Storm需要通过它来使用ZMQ。

进入解压后的jzmq-master文件夹中，依次运行以下命令：

```
./autogen.sh
./configure
make
sudo make install
```

如果运行make命令的过程中发生以下错误：

```
*** No rule to make target `classdist_noinst.stamp', needed by `org/zeromq/ZMQ.class'.  Stop.
```

则执行以下操作。

首先，执行以下命令：

```
touch src/classdist_noinst.stamp
```

接着进入src/org/zeromq文件夹中执行javac *.java这条命令，最后回退到jzmq-master文件夹的

根目录下，依次执行：

```
make
sudo make install
```

(6) 启动ZooKeeper

进入解压后的zookeeper-3.4.5文件夹中，将文件./conf/zoo_sample.cfg重命名为./conf/zoo.cfg。运行以下命令启动ZooKeeper：

```
bin/zkServer.sh start
```

然后检查ZooKeeper是否成功启动，此时先执行如下命令：

```
bin/zkCli.sh -server 127.0.0.1:2181
```

此时会出现一个交互窗口，在其中运行 `ls /`。

(7) 启动Storm

进入解压后的storm-0.9.0-rc2文件夹的bin目录中，依次执行以下命令启动Nimbus、Supervisor以及UI：

```
./storm nimbus
./storm supervisor
./storm ui
```

等待命令都执行完成后，打开链接http://localhost:8080，此时应该能看到Storm UI界面。

(8) 编译storm-stater jar包

接下来，我们在运行起来的集群上提交一个Topology，这里我们使用storm-starter做示范。

如果没有安装过git工具，可以运行下面的命令安装：

```
sudo apt-get install git
```

如果没有安装过leiningen工具，则按照https://github.com/technomancy/leiningen的步骤安装。

我们把下载下来的storm-starter源代码保存在~/project/storm-starter中。首先，进入该目录中：

```
cd ~/project/storm-starter
```

从GitHub上克隆一份storm-starter的源代码：

```
git clone git://github.com/nathanmarz/storm-starter.git
```

依次执行以下命令创建项目jar包：

```
lein deps
lein compile
lein install
```

创建好的jar包storm-starter-0.0.1-SNAPSHOT.jar位于target目录下。

(9) 提交Topology

进入storm-0.9.0-rc2文件夹中的bin目录中，运行以下命令提交Topology：

```
./storm jar ~/project/storm-stater/target/storm-starter-0.0.1-SNAPSHOT.jar storm.starter.WordCount
    Topology wordcount
```

等待提交结束后，刷新页面http://localhost:8080，我们可以看到提交的“wordcount” Topology，点击wordcount可以看到其详细运行情况。

至此，我们已经成功地在本地搭建好Storm集群，并且成功地运行了我们的第一个Topology。

2.2　搭建多机 Storm 集群

我们可以用多台机器搭建一个多机运行环境，这也是Storm的实际应用场景，下面我们来看一下怎么搭建这样的集群。

假设我们有3台机器M1、M2以及M3，它们的IP地址分别为10.1.172.1、10.1.172.2以及10.1.172.3，具体的分配情况如下：

- M1作为Nimbus
- M2作为ZooKeeper
- M3作为Supervisor

2.2.1　设置环境

首先，我们需要按照搭建单机集群的方式在每台机器上都搭建好Storm环境，也即执行2.1节的前5步。

接下来，修改M1和M3上的storm.yaml文件，它的路径是~/project/Storm/storm-0.9.0-rc2/conf/。修改storm.yaml的内容为：

```
java.library.path: "/usr/local/lib:/usr/lib:/opt/local/lib"
storm.zookeeper.servers:
    - "10.1.172.2"
nimbus.host: "10.1.172.1"
ui.port: 83
supervisor.slots.ports:
    - 6700
    - 6701
    - 6702
    - 6703
```

下面解释一下这些配置项。

- `java.library.path`：该配置项配置启动Storm所需lib包的路径。
- `storm.zookeeper.servers`：该配置项配置了当前集群中所有ZooKeeper机器的IP地址。我们只有一个ZooKeeper服务器，所以只配置了一个IP。
- `nimbus.host`：该配置项指明了Nimbus机器的IP地址。

- `ui.port`：该配置项配置了Storm UI使用的端口。如果不配置该项，默认使用8080端口，这里设置为使用83端口。
- `supervisor.slots.ports`：该配置项指明了一台Supervisor机器上所有可以使用的slot信息，也即端口号。表明该机器上最多可以启动4个Worker。

Storm还提供了很多其他配置项，我们会在最后一章中对这些常见配置项进行详细的介绍。

2.2.2　启动Storm集群

首先，在M2上启动ZooKeeper服务，并验证它是否成功启动。

接下来，在M1上进入storm-0.9.0-rc2文件夹的bin目录，依次执行以下命令启动Nimbus和UI：

```
./storm nimbus
./storm ui
```

最后在M3上进入storm-0.9.0-rc2文件夹的bin目录中，执行以下命令启动Supervisor：

```
./storm supervisor
```

等待都启动完成后，访问http://10.1.172.1:83就能看到启动起来的集群。

2.2.3　提交Topology

在M1（也即Nimbus所在的机器）上将`WordCountTopology`提交到集群中，然后刷新一下http://10.1.172.1:83页面，就能看到提交的Topology了。

至此，我们就完成了部署一个简单多机Storm 集群并提交Topology到集群运行的所有步骤。接下来，我们借助`WordCountTopology`示例，介绍一下Topology的组成。

2.3　WordCountTopology 介绍

`WordCountTopology`是一个基本的Storm Topology，由三个组件构成：

- `RandomSentenceSpout`
- `SplitSentence`
- `WordCount`

下面分别介绍这几个组件。

2.3.1　RandomSentenceSpout

这个类定义了一个Spout，它继承自`BaseRichSpout`。`BaseRichSpout`是一个实现了`IRichBolt`接口的虚类，这个接口是Storm中的一个主要接口。它的`nextTuple`方法随机地从一个句子数组中选出一个句子发送出去，`declareOutputFields`方法声明了该Spout输出的消息模式，这里输出只有一列，字段名是`word`：

```
public class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;

    @Override
    public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
        _collector = collector;
        _rand = new Random();
    }

    @Override
    public void nextTuple() {
        Utils.sleep(100);
        String[] sentences = new String[] {
            "the cow jumped over the moon",
            "an apple a day keeps the doctor away",
            "four score and seven years ago",
            "snow white and the seven dwarfs",
            "i am at two with nature"};
        String sentence = sentences[_rand.nextInt(sentences.length)];
        _collector.emit(new Values(sentence));
    }

    @Override
    public void ack(Object id) {
    }

    @Override
    public void fail(Object id) {
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```

2.3.2　SplitSentence

该类定义了一个Bolt，它继承自BaseBasicBolt。BaseBasicBolt是一个实现了IBasicBolt接口的虚类。execute方法是Bolt真正处理业务逻辑的地方，它将从Spout收到的句子按照空格分割，然后把每一个单词作为一条信息发送出去。declareOutputFields方法声明该Bolt的输出消息格式，这里也只有一列，字段名是word。SplitSentence类的定义如下：

```
public static class SplitSentence extends BaseBasicBolt {
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String sentence = tuple.getString(0);
        for (String word : sentence.split(" "))
            collector.emit(new Values(word));
    }
```

```
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```


2.3.3 WordCount

类WordCount跟SplitSentence类似，也定义了一个Bolt。这个类对收到的所有单词进行计数统计，execute方法更新收到单词的缓存数，并将当前该单词及其对应的数目发送出去；declareOutputFields方法声明该Bolt的输出消息格式，这里输出有两列，字段名分别是word和count；cleanup方法在该Topology被停掉的时候被调用（不保证一定能够调用到），它将当前缓存的所有单词及数目信息打印到日志中。类WordCount的定义如下：

```
public static class WordCount extends BaseBasicBolt {
    Map<String, Integer> counts = new HashMap<String, Integer>();

    @Override
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String word = tuple.getString(0);
        Integer count = counts.get(word);
        if(count==null) count = 0;
        count++;
        counts.put(word, count);
        collector.emit(new Values(word, count));
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }

    @Override
    public void cleanup(){
        for(Map.Entry<String, Integer> entry : counts.entrySet()){
            logger.info(entry.getKey() + ": " + entry.getValue());
        }
    }
}
```

2.3.4 WordCountTopology构建

类WordCountTopology是真正定义Topology的地方，其代码如下所示：

```
1 public class WordCountTopology {
2     public static void main(String[] args) throws Exception {
3         TopologyBuilder builder = new TopologyBuilder();
4         builder.setSpout("spout", new RandomSentenceSpout(), 5);
5         builder.setBolt("split", new SplitSentence(), 8)
```

```
6               .shuffleGrouping("spout");
7           builder.setBolt("count", new WordCount(), 12)
8               .fieldsGrouping("split", new Fields("word"));
9
10          Config conf = new Config();
11          conf.setDebug(true);
12
13          if(args!=null && args.length > 0) {
14              conf.setNumWorkers(3);
15              StormSubmitter.submitTopology("WordCountTopology", conf, builder.createTopology());
16          } else {
17              conf.setMaxTaskParallelism(3);
18
19              LocalCluster cluster = new LocalCluster();
20              cluster.submitTopology("WordCountTopology", conf, builder.createTopology());
21              Thread.sleep(10000);
22              cluster.shutdown();
23          }
24      }
25  }
```

- 第3行创建一个TopologyBuilder对象，这个类是用来构建基本Topology的，后面我们会详细介绍它。
- 第4行设置Topology的Spout，它的id是spout，这里创建一个RandomSentenceSpout对象作为Spout对象，并行度设置为5。
- 第5~6行设置Topology的Bolt，它的id是split，这里创建一个SplitSentence对象作为Bolt对象，并行度设置为8。它接收spout发出的消息，其分组策略是随机分组（Shuffle Grouping），即spout的多个实例会随机分发消息到split的各个实例上。
- 第7~8行设置Topology的另一个Bolt，它的id是count，这里创建一个WordCount对象作为Bolt对象，并行度设置为12。它接收split发出的消息，其分组策略是域分组（Fields Grouping），即split的各个实例会按照消息word列所对应的值决定将消息发送到count的哪个实例中。所有word列值相同的消息会被发到同一个count节点中处理。
- 第10~11行设置该Topology所用的配置信息，这里仅设置了调试模式为真。这样系统会打印所有发送及接收的消息。
- 第13~15行是在集群上提交该Topology，我们前面介绍的单机跟多机环境都是用这种方式运行的。
- 第17~22行是直接运行该Topology，但它不会提交到真实的集群上。Storm提供了一个LocalCluster对象来模拟集群运行环境，它采用线程模拟进程的方式实现，一般用于调试写好的Topology。

第 3 章

Storm编程基础

本章将介绍一些Storm中的基础概念，以及相关类和接口的用法。另外，在网站GitHub的wiki页面上（https://github.com/ nathanmarz/storm/wiki/Concepts），还专门提供了有关Storm核心概念的简要描述，读者可以将其与本章的讲解对照起来进行学习。

3.1 Fields 定义

Fields数据结构用于存储消息的字段名列表，其所需参数是字段名集合。对于同一条消息，在构建Fields对象时会为其所有的字段建立索引。它的定义如下：

```
1 public class Fields implements Iterable<String>, Serializable {
2     private List<String> _fields;
3     private Map<String, Integer> _index = new HashMap<String, Integer>();
4
5     public Fields(String... fields) {
6         this(Arrays.asList(fields));
7     }
8
9     public Fields(List<String> fields) {
10        _fields = new ArrayList<String>(fields.size());
11        for (String field : fields) {
12            if (_fields.contains(field))
13                throw new IllegalArgumentException(
14                    String.format("duplicate field '%s'", field)
15                );
16            _fields.add(field);
17        }
18        index();
19    }
20    ...
21 }
```

- 第1行表明Fields类实现了接口Iterable<String>和Serializable。接口Iterable<String>定义了一个迭代器接口，用于遍历Fields中存储的字段名列表；接口Serializable则表明该类是可以被序列化的。
- 第2~3行分别定义了一个保存所有字段名的列表，以及一个保存了从字段名到它在字段名

列表中位置的映射表。

- 第5~7行的构造函数接收一个可变参数fields（也即一个字段名数组），将fields转换为列表后调用第9~19行定义的构造函数。
- 第9~19行定义的构造函数会首先检查传入的字段名列表中的字段名是否有重复，并保存该字段名列表，最后调用index方法为该字段名列表建立索引。

index方法实际上就是遍历字段名列表，将每个字段名和它对应的位置保存到第3行定义的映射表中，其代码如下：

```
1 private void index() {
2     for(int i=0; i<_fields.size(); i++) {
3         _index.put(_fields.get(i), i);
4     }
5 }
```

此外，Fields数据结构中还定义了很多常用方法，比如获取所有字段名、获取字段名列表的大小、获取指定位置的字段名、获取某个字段名的索引位置以及获取一个所有字段名的迭代器等。这些方法都相对简单易懂，限于篇幅，此处不再赘述。

3.2　Tuple 接口

Tuple是Storm中的主要数据结构。在Storm发送接收消息的过程中，每一条消息实际上都是一个Tuple对象。下面首先来看一下Tuple接口的定义：

```
1 public interface Tuple {
2     public int size();
3     public int fieldIndex(String field);
4     public boolean contains(String field);
5 
6     public Object getValue(int i);
7     public String getString(int i);
8     public Integer getInteger(int i);
9     public Long getLong(int i);
10    public Boolean getBoolean(int i);
11    public Short getShort(int i);
12    public Byte getByte(int i);
13    public Double getDouble(int i);
14    public Float getFloat(int i);
15    public byte[] getBinary(int i);
16 
17    public Object getValueByField(String field);
18    public String getStringByField(String field);
19    public Integer getIntegerByField(String field);
20    public Long getLongByField(String field);
21    public Boolean getBooleanByField(String field);
22    public Short getShortByField(String field);
23    public Byte getByteByField(String field);
24    public Double getDoubleByField(String field);
```

```
25      public Float getFloatByField(String field);
26      public byte[] getBinaryByField(String field);
27
28      public List<Object> getValues();
29      public Fields getFields();
30      public List<Object> select(Fields selector);
31
32      public GlobalStreamId getSourceGlobalStreamid();
33      public String getSourceComponent();
34      public int getSourceTask();
35      public String getSourceStreamId();
36      public MessageId getMessageId();
37  }
```


- 第2行的size方法返回当前消息中字段的数目。
- 第3行的fieldIndex方法可根据传入的字段名获取该字段在所有字段中所处的位置。
- 第4行的contains方法用来判断该消息是否包含指定的字段。
- 第6~15行的方法用于获取由参数i指定的字段位置的值。如果用户知道该字段对应的类型，就可调用对应类型的获取方法获取字段的值。若字段的类型跟获取方法不匹配，将发生异常。
- 第17~26行跟第6~15行类似，只不过这里的方法是根据字段名获取对应的值。定义这些重载方法的目的都是为了提高消息处理的性能。
- 第28行的getValues方法用来获取该消息存储的值列表。
- 第29行的getFields方法用来获取存储了所有字段名的Fields对象。
- 第30行的select方法用来获取由参数Fields指定的与字段名对应的值列表。
- 第32行用来获取与该消息对应的GlobalStreamId，后面我们会介绍这个类。
- 第33行用来获取创建这个消息的组件id。
- 第34行用来获取创建这个消息的TaskId，第11章将专门介绍Task。
- 第35行用来获取该消息被发送到的流的序号。
- 第36行用来获取该Tuple的消息序号，该序号会被Storm用来追踪消息是否处理成功，详细内容请参考第12章。

Storm提供了Tuple的默认实现类TupleImpl。它除了实现Tuple接口之外，还实现了Clojure定义的几个接口Seqable、Indexed和IMeta，实现这些接口的目的是为了在Clojure代码中能更好地操纵Tuple对象。TupleImpl的实现比较容易理解，用户可以自行查看代码，这里不再赘述。

3.3 常用声明接口

Storm中有多个与组件声明相关的类，它们的主要作用是让用户更加方便地定义组件的输入输出，以及一些与组件相关的配置，其类关系如图3-1所示。

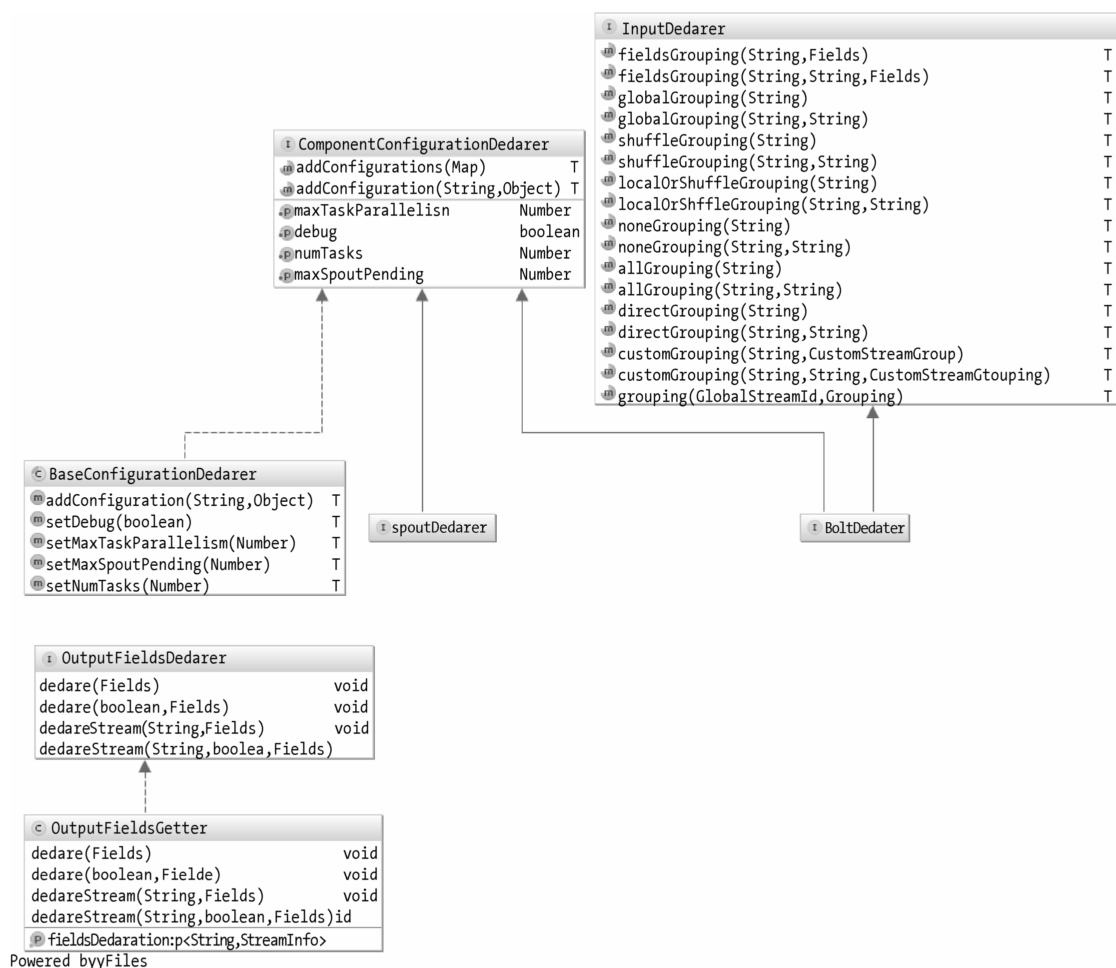


图3-1 常用声明接口

3.3.1 配置声明接口

ComponentConfigurationDeclarer接口定义了一些和组件相关的配置项。该接口中定义的方法返回值为通用类型T，且T实现了ComponentConfigurationDeclarer。由于该接口中的这些方法返回的是相同的对象，因此在用法上可以实现方法的级联。该接口的定义如下。

```
public interface ComponentConfigurationDeclarer<T extends ComponentConfigurationDeclarer> {
    T addConfigurations(Map conf);
    T addConfiguration(String config, Object value);
    T setDebug(boolean debug);
    T setMaxTaskParallelism(Number val);
    T setMaxSpoutPending(Number val);
```

```
    T setNumTasks(Number val);
}
```

Storm默认提供了一个抽象类BaseConfigurationDeclarer，它实现了以上接口中除addConfigurations(Map conf)以外的大部分方法：

```
public abstract class BaseConfigurationDeclarer<T extends ComponentConfigurationDeclarer> implements
    ComponentConfigurationDeclarer<T> {
    @Override
    public T addConfiguration(String config, Object value) {
        Map configMap = new HashMap();
        configMap.put(config, value);
        return addConfigurations(configMap);
    }

    @Override
    public T setDebug(boolean debug) {
        return addConfiguration(Config.TOPOLOGY_DEBUG, debug);
    }

    @Override
    public T setMaxTaskParallelism(Number val) {
        if(val!=null) val = val.intValue();
        return addConfiguration(Config.TOPOLOGY_MAX_TASK_PARALLELISM, val);
    }

    @Override
    public T setMaxSpoutPending(Number val) {
        if(val!=null) val = val.intValue();
        return addConfiguration(Config.TOPOLOGY_MAX_SPOUT_PENDING, val);
    }

    @Override
    public T setNumTasks(Number val) {
        if(val!=null) val = val.intValue();
        return addConfiguration(Config.TOPOLOGY_TASKS, val);
    }
}
```

3.3.2 输入声明接口

接口InputDeclarer采用了类似ComponentConfigurationDeclarer的定义方式，即可以级联使用，用于声明一个组件的输入，定义如下：

```
public interface InputDeclarer<T extends InputDeclarer> {
    public T fieldsGrouping(String componentId, Fields fields);
    public T fieldsGrouping(String componentId, String streamId, Fields fields);

    public T globalGrouping(String componentId);
    public T globalGrouping(String componentId, String streamId);
```

```
    public T shuffleGrouping(String componentId);
    public T shuffleGrouping(String componentId, String streamId);

    public T localOrShuffleGrouping(String componentId);
    public T localOrShuffleGrouping(String componentId, String streamId);

    public T noneGrouping(String componentId);
    public T noneGrouping(String componentId, String streamId);

    public T allGrouping(String componentId);
    public T allGrouping(String componentId, String streamId);

    public T directGrouping(String componentId);
    public T directGrouping(String componentId, String streamId);

    public T customGrouping(String componentId, CustomStreamGrouping grouping);
    public T customGrouping(String componentId, String streamId, CustomStreamGrouping grouping);

    public T grouping(GlobalStreamId id, Grouping grouping);

}
```

其中定义了各种分组方式,每个方法的返回值均为类型T,即实际运行中实现了InputDeclarer的类型。如果有多个输入，这样定义之后就可以用级联的方式声明多个输入的分组方式。

如果不指定流序号，则表示默认在ID为default的流上进行分组。

该接口在创建Topology的时候会用到。

3.3.3　输出字段声明接口

接口OutputFieldsDeclarer定义了Topology中每个组件的输出字段声明。这个接口非常重要，每个Topology中的组件都需要用它来指定输出到哪些流、声明输出的字段列表以及指明输出流是否是直接流（Direct Stream）。它的定义如下：

```
1 public interface OutputFieldsDeclarer {
2    /**
3    * Uses default stream id.
4    */
5    public void declare(Fields fields);
6    public void declare(boolean direct, Fields fields);
7
8    public void declareStream(String streamId, Fields fields);
9    public void declareStream(String streamId, boolean direct, Fields fields);
10}
```

它使用了我们前面介绍过的Fields对象来声明输出字段列表。第5~6行的方法没有指明输出流号，默认使用的是default。第5行和第8行的方法没有指明是否为直接流，默认是不使用直接流的方式输出。本书的后续章节会介绍直接流方式输出的相关概念，以及何时使用直接流方式的输出。

3.3.4 组件声明接口

Storm中的组件包括Spout和Bolt，所以组件声明接口也有两种：Spout Dedarer和BoltDeclarer。

接口SpoutDeclarer仅仅是简单地继承了接口ComponentConfigurationDeclarer，用于声明一个组件以及设定配置项，其定义如下：

```
public interface SpoutDeclarer extends ComponentConfigurationDeclarer<SpoutDeclarer> {

}
```

接口BoltDeclarer同时继承了接口ComponentConfigurationDeclarer以及接口InputDeclarer：

```
public interface BoltDeclarer extends InputDeclarer<BoltDeclarer>, ComponentConfigurationDeclarer
    BoltDeclarer> {

}
```

从用户的角度来看，Spout表示了消息的源头，即Spout是没有输入的，而Bolt是中间的处理节点，它需要定义它的输入，即需要实现InputDeclarer接口。

3.4 Spout 输出收集器

Storm只定义了一个Spout输出收集器的接口ISpoutOutputCollector，并提供了它的一个默认实现SpoutOutputCollector。在Executor中，我们提供了ISpoutOutputCollector接口的真正实现。下面我们分别予以介绍。

3.4.1 ISpoutOutputCollector和SpoutOutputCollector

首先看一下ISpoutOutputCollector接口的定义：

```
public interface ISpoutOutputCollector {
    /**
        Returns the task ids that received the tuples.
    */
    List<Integer> emit(String streamId, List<Object> tuple, Object messageId);
    void emitDirect(int taskId, String streamId, List<Object> tuple, Object messageId);
    void reportError(Throwable error);
}
```

- ❑ emit方法用来向外发送数据，它的返回值是该消息所有发送目标的TaskId集合，其输入参数的含义如下。
    - ■ streamId：消息将被输出到的流。
    - ■ tuple：要输出的消息，为一个Object列表。
    - ■ messageId：输出消息的标记信息。如果messageId被设置为null，Storm将不会追踪该消息，否则它会被用来追踪所发出消息的处理情况，具体情况可参考第12章。

- emitDirect方法的输入列表与emit方法相似，主要区别在于使用emitDirect时，只有由参数taskId所指定的Task才可以接收这条消息。这个方法要求与参数streamId相对应的流必须被定义为直接流，同时接收端的Task也必须以直接分组（Direct Grouping）的方式来接收消息，否则会有异常抛出。另外，如果没有下游节点接收该消息，那么该消息其实也就没有被真正发送。
- reportError方法用来处理异常。

接下来，我们看一下Storm提供的该接口的默认实现类SpoutOutputCollector。这个类实际上是一个代理类，它本身也封装了一个ISpoutOutputCollector对象，所有的操作实际上都是通过该对象来实现的。除此之外，它还提供了一些重载方法以方便用户使用。该类的定义如下所示：

```
1 public class SpoutOutputCollector implements ISpoutOutputCollector {
2     ISpoutOutputCollector _delegate;
3
4     public SpoutOutputCollector(ISpoutOutputCollector delegate) {
5         _delegate = delegate;
6     }
7
8     public List<Integer> emit(String streamId, List<Object> tuple, Object messageId) {
9         return _delegate.emit(streamId, tuple, messageId);
10    }
11
12    public List<Integer> emit(List<Object> tuple, Object messageId) {
13        return emit(Utils.DEFAULT_STREAM_ID, tuple, messageId);
14    }
15
16    public List<Integer> emit(List<Object> tuple) {
17        return emit(tuple, null);
18    }
19
20    public List<Integer> emit(String streamId, List<Object> tuple) {
21        return emit(streamId, tuple, null);
22    }
23
24    public void emitDirect(int taskId, String streamId, List<Object> tuple, Object messageId) {
25        _delegate.emitDirect(taskId, streamId, tuple, messageId);
26    }
27
28    public void emitDirect(int taskId, List<Object> tuple, Object messageId) {
29        emitDirect(taskId, Utils.DEFAULT_STREAM_ID, tuple, messageId);
30    }
31
32    public void emitDirect(int taskId, String streamId, List<Object> tuple) {
33        emitDirect(taskId, streamId, tuple, null);
34    }
35
36    public void emitDirect(int taskId, List<Object> tuple) {
37        emitDirect(taskId, tuple, null);
38    }
39
```

```
40    @Override
41    public void reportError(Throwable error) {
42        _delegate.reportError(error);
43    }
44}
```

- 第2行定义了它所代理的ISpoutOutputCollector对象。
- 第4~6行定义构造函数，它接收一个ISpoutOutputCollector对象，并将其保存在第2行定义的变量中。
- 第8~22行实现了接口中的emit函数，并且提供了它的几个重载方法。比如，如果不指定streamId，默认使用default；如果不指定messageId，则默认使用空（null）。
- 第24~38行实现了接口中的emitDirect函数，同时也提供了几个重载方法，这跟emit函数是一致的。
- 第41~43行定义了reportError的实现。

3.4.2 Executor中ISpoutOutputCollector的实现

Executor在调用Spout的prepare方法时也提供了一种ISpoutOutputCollector接口的实现。以下代码在文件executor.clj中：

```
1 (SpoutOutputCollector.
2     (reify ISpoutOutputCollector
3     (^List emit [this ^String stream-id ^List tuple ^Object message-id]
4         (send-spout-msg stream-id tuple message-id nil)
5         )
6     (^void emitDirect [this ^int out-task-id ^String stream-id
7         ^List tuple ^Object message-id]
8         (send-spout-msg stream-id tuple message-id out-task-id)
9         )
10    (reportError [this error]
11       (report-error error)
12       )))
```

- 第2~12行定义了ISpoutOutputCollector的实现。第1行初始化了一个SpoutOutputCollector对象，用户代码通过这个对象向外发送数据，Storm中所有Spout的输出收集器归根到底都是使用该对象向外发送数据的。reify是Clojure的关键字，它用于实现一个接口，并初始化一个对象。
- 第3~5行定义了emit方法，它通过调用函数send-spout-msg来发送数据，该函数会在第10章中详细介绍。
- 第6~9行定义了emitDirect方法，它同emit方法类似，也是通过调用函数send-spout-msg来发送数据的。
- 第10~12行定义了reportError方法，它是通过调用report-error方法实现的，该方法也将在第10章中介绍。

3.5 Bolt 输出收集器

用户在实现Bolt时要明确：Bolt处理好的消息都是通过输出收集器发送出去的，不同类型的Bolt所使用的输出收集器也是不同的，下面简要介绍一下。

- IRichBolt：它使用OutputCollector输出收集器，该收集器实现的是IOutputCollector接口，实际上是一个代理类。
- IBasicBolt：它使用BasicOutputCollector输出收集器，该收集器实际上是OutputCollector的封装类，实现的是IBasicOutputCollector接口。
- IBatchBolt：它使用BatchOutputCollector输出收集器，该收集器是一个虚基类，Storm提供了它的默认实现类BatchOutputCollectorImpl，这个类实际上也是通过封装OutputCollector类来实现消息发送的。

下面我们分别对这几种输出收集器进行介绍。

3.5.1 IOutputCollector和OutputCollector

接口IErrorReporter定义了reportError方法，其输入为一个Throwable对象，用户可以在该方法中处理异常：

```
public interface IErrorReporter {
    void reportError(Throwable error);
}
```

接口IOutputCollector扩展了接口IErrorReporter，并且定义了一些基本方法：

```
public interface IOutputCollector extends IErrorReporter {
    /**
    *  Returns the task ids that received the tuples.
    */
    List<Integer> emit(String streamId, Collection<Tuple> anchors, List<Object> tuple);
    void emitDirect(int taskId, String streamId, Collection<Tuple> anchors, List<Object> tuple);
    void ack(Tuple input);
    void fail(Tuple input);
}
```

- emit方法用来向外发送数据，它的返回值是该消息所有发送目标的TaskId集合，其输入参数的含义如下所示。
  - streamId：消息将被输出到的流。
  - anchors：输出消息的标记，通常代表该条消息是由哪些消息产生的，主要用于消息的Ack系统。
  - tuple：要输出的消息，为一个Object列表。
- emitDirect方法的输入列表与emit方法相似，主要区别在于，emitDirect发送的消息只有指定的Task才可以接收。这个方法要求streamId对应的流必须被定义为直接流，同时接收

端的Task必须通过直接分组的方式来接收消息，否则会抛出异常。如果没有下游节点接收该消息，那么此类消息其实也就没有被真正发送。

- fail和ack方法用来表示消息是否被成功处理。

Storm提供了IOutputCollector接口的默认实现类OutputCollector，它实际上也是一个代理。它包含一个真正工作的IOutputCollector实例，这个对象是在Clojure代码中定义的。OutputCollector主要用于从IRichBolt向外发送数据。在OutputCollector的实现中，所有操作都由代理对象完成。而且，OutputCollector还提供了很多重载的方法以方便用户使用，其定义如下：

3

```
1 /**
2 * This output collector exposes the API for emitting tuples from an IRichBolt.
3 * This is the core API for emitting tuples. For a simpler API, and a more restricted
4 * form of stream processing, see IBasicBolt and BasicOutputCollector.
5 */
6 public class OutputCollector implements IOutputCollector {
7     private IOutputCollector _delegate;
8
9
10 public OutputCollector(IOutputCollector delegate) {
11       _delegate = delegate;
12 }
13
14 /**
15   * Emits a new tuple to a specific stream with a single anchor. The emitted values must be
16   * immutable.
17   *
18   * @param streamId the stream to emit to
19   * @param anchor the tuple to anchor to
20   * @param tuple the new output tuple from this bolt
21   * @return the list of task ids that this new tuple was sent to
22   */
23 public List<Integer> emit(String streamId, Tuple anchor, List<Object> tuple) {
24     return emit(streamId, Arrays.asList(anchor), tuple);
25 }
26
27 public List<Integer> emit(String streamId, List<Object> tuple) {
28     return emit(streamId, (List) null, tuple);
29 }
30
31 public List<Integer> emit(Collection<Tuple> anchors, List<Object> tuple) {
32    return emit(Utils.DEFAULT_STREAM_ID, anchors, tuple);
33 }
34
35
36 public List<Integer> emit(Tuple anchor, List<Object> tuple) {
37     return emit(Utils.DEFAULT_STREAM_ID, anchor, tuple);
38 }
39
40 public List<Integer> emit(List<Object> tuple) {
41    return emit(Utils.DEFAULT_STREAM_ID, tuple);
42 }
```

```
43
44 /**
45   * Emits a tuple directly to the specified task id on the specified stream.
46   * If the target bolt does not subscribe to this bolt using a direct grouping,
47   * the tuple will not be sent. If the specified output stream is not declared
48   * as direct, or the target bolt subscribes with a non-direct grouping,
49   * an error will occur at runtime. The emitted values must be
50   * immutable.
51   *
52   * @param taskId the taskId to send the new tuple to
53   * @param streamId the stream to send the tuple on. It must be declared as a direct stream in
        the topology definition.
54   * @param anchor the tuple to anchor to
55   * @param tuple the new output tuple from this bolt
56   */
57 public void emitDirect(int taskId, String streamId, Tuple anchor, List<Object> tuple) {
58     emitDirect(taskId, streamId, Arrays.asList(anchor), tuple);
59 }
60
61 public void emitDirect(int taskId, String streamId, List<Object> tuple) {
62     emitDirect(taskId, streamId, (List) null, tuple);
63 }
64
65 public void emitDirect(int taskId, Collection<Tuple> anchors, List<Object> tuple) {
66     emitDirect(taskId, Utils.DEFAULT_STREAM_ID, anchors, tuple);
67 }
68
69 public void emitDirect(int taskId, Tuple anchor, List<Object> tuple) {
70     emitDirect(taskId, Utils.DEFAULT_STREAM_ID, anchor, tuple);
71 }
72
73
74 public void emitDirect(int taskId, List<Object> tuple) {
75     emitDirect(taskId, Utils.DEFAULT_STREAM_ID, tuple);
76 }
77
78 @Override
79 public List<Integer> emit(String streamId, Collection<Tuple> anchors, List<Object> tuple) {
80     return _delegate.emit(streamId, anchors, tuple);
81 }
82
83 @Override
84 public void emitDirect(int taskId, String streamId, Collection<Tuple> anchors, List<Object> tuple){
85     _delegate.emitDirect(taskId, streamId, anchors, tuple);
86 }
87
88 @Override
89 public void ack(Tuple input) {
90     _delegate.ack(input);
91 }
92
93 @Override
94 public void fail(Tuple input) {
95     _delegate.fail(input);
```

```
96 }
97
98 @Override
99 public void reportError(Throwable error) {
100     _delegate.reportError(error);
101 }
```

- 第10~12行为类OutputCollector的构造函数，它需要传入一个实现了IOutputCollector的对象。
- 第23~42行定义了emit的各种重载方法，具体如下所示。
    - 若未传入streamId，使用default作为流号。
    - 若未传入anchor，使用对象null作为标记。
    - 若传入的anchor不是列表对象，将其转化成为列表对象。
- 第44~76行定义了emtiDirect的各种重载方法，它与emit方法很类似。
- 第78~101行实现了IOutputCollector接口，利用代理对象实现这些方法。

3.5.2　IBasicOutputCollector和BasicOutputCollector

我们看一下接口IBasicOutputCollector的定义：

```
public interface IBasicOutputCollector {
    List<Integer> emit(String streamId, List<Object> tuple);
    void emitDirect(int taskId, String streamId, List<Object> tuple);
    void reportError(Throwable t);
}
```

首先介绍一下为什么会有IBasicOutputCollector。这个接口是在IBasicBolt中使用的，对比IOutputCollector可以看出它们的区别。

- IBasicOutputCollector没有Ack和Fail方法。
- IBasicOutputCollector的emit和emitDirect方法中没有anchor参数。

这样设计的原因是如果使用IBasicBolt，Storm框架会自动帮用户进行Ack、Fail和Anchor操作，用户自己不需要关心这一点，后面介绍IBasicBolt的时候我们会详细介绍这一点。所以为了确保这种机制正常运行，避免用户在使用时出错，Storm提供了简化版的IBasicOutputCollector。当然，简化也就意味着使用IBasicBolt是有限制的，后面也会介绍这一点。

BasicOutputCollector是Storm提供的IBasicOutputCollector接口的默认实现，其中包含了一个OutputCollector类型的成员变量，实际上所有的消息最终都将由这个OutputCollector进行处理。BasicOutputCollector还提供了一些简易的方法进行消息标记。下面简单分析如下：

```
1 public class BasicOutputCollector implements IBasicOutputCollector {
2     private OutputCollector out;
3     private Tuple inputTuple;
4
5     public BasicOutputCollector(OutputCollector out) {
6         this.out = out;
```

```
7     }
8
9     public List<Integer> emit(String streamId, List<Object> tuple) {
10        return out.emit(streamId, inputTuple, tuple);
11    }
12
13    public List<Integer> emit(List<Object> tuple) {
14        return emit(Utils.DEFAULT_STREAM_ID, tuple);
15    }
16
17    public void setContext(Tuple inputTuple) {
18        this.inputTuple = inputTuple;
19    }
20
21    public void emitDirect(int taskId, String streamId, List<Object> tuple) {
22        out.emitDirect(taskId, streamId, inputTuple, tuple);
23    }
24
25    public void emitDirect(int taskId, List<Object> tuple) {
26        emitDirect(taskId, Utils.DEFAULT_STREAM_ID, tuple);
27    }
28
29    protected IOutputCollector getOutputter() {
30        return out;
31    }
32
33    public void reportError(Throwable t) {
34        out.reportError(t);
35    }
36 }
```

- 第2行定义代理的out对象，类型为OutputCollector。
- 第3行定义输入的消息，它将作为消息的标记对象，用于Ack系统。
- 第9~15行定义emit方法以及重载，若未指定streamId，将发送到default流。
- 第17~19行定义setContext方法，将输入的消息作为上下文。若未重新调用该函数，则接下来由emit发送出去的消息都以该消息作为标记，也即都是由该消息衍生出来的。
- 第21~27行定义emitDirect方法及重载，若未指定streamId，将发送到default流。

3.5.3　BatchOutputCollector和BatchOutputCollectorImpl

BatchOutputCollector是Storm中用于数据批处理的输出收集器，它的定义如下：

```
public abstract class BatchOutputCollector {

    /**
    * Emits a tuple to the default output stream.
    */
    public List<Integer> emit(List<Object> tuple) {
        return emit(Utils.DEFAULT_STREAM_ID, tuple);
    }
```

```
    public abstract List<Integer> emit(String streamId, List<Object> tuple);

    /**
    * Emits a tuple to the specified task on the default output stream. This output
    * stream must have been declared as a direct stream, and the specified task must
    * use a direct grouping on this stream to receive the message.
    */
    public void emitDirect(int taskId, List<Object> tuple) {
        emitDirect(taskId, Utils.DEFAULT_STREAM_ID, tuple);
    }

    public abstract void emitDirect(int taskId, String streamId, List<Object> tuple);

    public abstract void reportError(Throwable error);
}
```

可以看出，它的方法跟IBasicOutputCollector中定义的接口方法基本一致，这里不再赘述。而且，它的设计思路也跟IBasicOutputCollector类似，常用在IBatchBolt中，不需要自己去处理Ack、Fail和Anchor这3项操作，Storm框架默认实现了这些操作。因此在IBatchBolt中，用户只需要关心数据发送和错误处理即可。

Storm提供了BatchOutputCollector的默认实现类BatchOutputCollectorImpl，它实际上是一个代理类，内部封装了OutputCollector变量，所有的方法都通过调用OutputCollector的方法来实现，其代码如下：

```
public class BatchOutputCollectorImpl extends BatchOutputCollector {
    OutputCollector _collector;

    public BatchOutputCollectorImpl(OutputCollector collector) {
        _collector = collector;
    }

    @Override
    public List<Integer> emit(String streamId, List<Object> tuple) {
        return _collector.emit(streamId, tuple);
    }

    @Override
    public void emitDirect(int taskId, String streamId, List<Object> tuple) {
        _collector.emitDirect(taskId, streamId, tuple);
    }

    @Override
    public void reportError(Throwable error) {
        _collector.reportError(error);
    }

    public void ack(Tuple tup) {
        _collector.ack(tup);
    }
```

```
    public void fail(Tuple tup) {
        _collector.fail(tup);
    }
}
```

注意这里有ack和fail两个方法的实现，它们会在BatchBoltExecutor（3.8.4节将介绍）中用到。

3.5.4　Executor中的IOutputCollector实现

Executor在调用Bolt的prepare方法时实现了IOutputCollector。以下代码来源于executor.clj文件：

```
1 (OutputCollector.
2     (reify IOutputCollector
3         (emit [this stream anchors values]
4             (bolt-emit stream anchors values nil))
5         (emitDirect [this task stream anchors values]
6             (bolt-emit stream anchors values task))
7         (^void ack [this ^Tuple tuple]
8             (let [^TupleImpl tuple tuple
9                     ack-val (.getAckVal tuple)]
10                (fast-map-iter [[root id] (.. tuple getMessageId getAnchorsToIds)]
11                    (task/send-unanchored task-data
12                        ACKER-ACK-STREAM-ID
13                    [root (bit-xor id ack-val)])
14             ))
15            (let [delta (tuple-time-delta! tuple)]
16        (task/apply-hooks user-context .boltAck (BoltAckInfo. tuple task-id delta))
17        (when delta
18            (builtin-metrics/bolt-acked-tuple! (:builtin-metrics task-data)
19                executor-stats
20                (.getSourceComponent tuple)
21                (.getSourceStreamId tuple)
22                delta)
23            (stats/bolt-acked-tuple! executor-stats
24                (.getSourceComponent tuple)
25                (.getSourceStreamId tuple)
26                delta))))
27        (^void fail [this ^Tuple tuple]
28            (fast-list-iter [root (.. tuple getMessageId getAnchors)]
29               (task/send-unanchored task-data
30                  ACKER-FAIL-STREAM-ID
31                  [root]))
32            (let [delta (tuple-time-delta! tuple)]
33        (task/apply-hooks user-context .boltFail (BoltFailInfo. tuple task-id delta))
34        (when delta
35            (builtin-metrics/bolt-failed-tuple! (:builtin-metrics task-data)
36                executor-stats
37                (.getSourceComponent tuple)
38                (.getSourceStreamId tuple))
39            (stats/bolt-failed-tuple! executor-stats
40                (.getSourceComponent tuple)
41                (.getSourceStreamId tuple)
```

```
42              delta))))
43   (reportError [this error]
44       (report-error error)
45   )))
```

- 第2~45行实现了IOutputCollector接口。第1行则初始化了一个OutputCollector对象，这个对象就是在通常的用户代码中看到的对象，用户代码以及前面介绍过的各种输出收集器归根到底都会使用该对象来向外发送数据。
- 第3~4行实现了emit方法，其数据目标的TaskId为nil。emit方法主要调用clojure的bolt-emit函数，该函数将在第10章中详细介绍。
- 第5~6行实现了emitDirect方法，它也是基于bolt-emit函数来实现的。

3.6　组件接口

组件接口IComponent定义了如下两个方法。

- declareOutputFields方法：用于定义组件的输出Schema。
- getComponentConfiguration方法：用来描述一些与组件相关的配置。

IComponent主要用于构建Topology，所有的Spout和Bolt也都会实现该接口，其代码如下。

```
/**
* Common methods for all possible components in a topology. This interface is used
* when defining topologies using the Java API.
*/
public interface IComponent extends Serializable {
    void declareOutputFields(OutputFieldsDeclarer declarer);
    Map<String, Object> getComponentConfiguration();
}
```

3.7　Spout 接口

Storm中与Spout相关的接口主要有ISpout和IRichSpout，图3-2描述了它们之间的类关系。

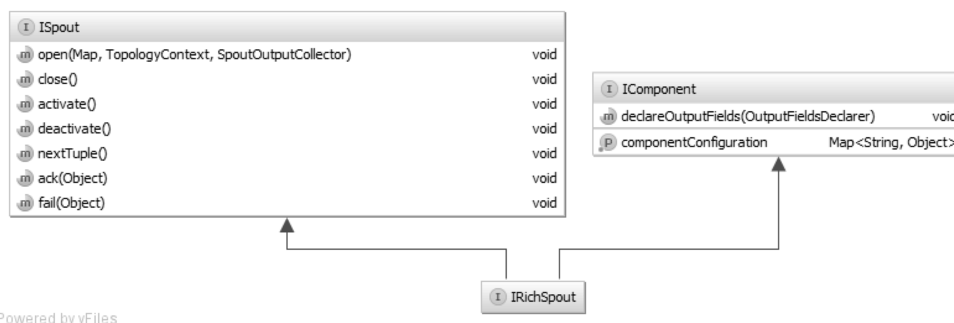


图3-2　Spout接口类关系图

3.7.1　ISpout

接口ISpout定义了作为Spout应该实现的功能集合：

```
/**
* ISpout is the core interface for implementing spouts. A Spout is responsible
* for feeding messages into the topology for processing. For every tuple emitted by
* a spout, Storm will track the (potentially very large) DAG of tuples generated
* based on a tuple emitted by the spout. When Storm detects that every tuple in
* that DAG has been successfully processed, it will send an ack message to the Spout.
*
* <p>If a tuple fails to be fully process within the configured timeout for the
* topology (see {@link backtype.storm.Config}), Storm will send a fail message to the spout
* for the message.</p>
*
* <p> When a Spout emits a tuple, it can tag the tuple with a message id. The message id
* can be any type. When Storm acks or fails a message, it will pass back to the
* spout the same message id to identify which tuple it's referring to. If the spout leaves out
* the message id, or sets it to null, then Storm will not track the message and the spout
* will not receive any ack or fail callbacks for the message.</p>
*
* <p>Storm executes ack, fail, and nextTuple all on the same thread. This means that an implementor
* of an ISpout does not need to worry about concurrency issues between those methods. However, it
* also means that an implementor must ensure that nextTuple is non-blocking: otherwise
* the method could block acks and fails that are pending to be processed.</p>
*/
public interface ISpout extends Serializable {
    /**
    * Called when a task for this component is initialized within a worker on the cluster.
    * It provides the spout with the environment in which the spout executes.
    *
    * <p>This includes the:</p>
    *
    * @param conf The Storm configuration for this spout. This is the configuration provided to the
        topology merged in with cluster configuration on this machine.
    * @param context This object can be used to get information about this task's place within the
        topology, including the task id and component id of this task, input and output information,etc.
    * @param collector The collector is used to emit tuples from this spout. Tuples can be emitted at
        any time, including the open and close methods. The collector is thread-safe and should be
        saved as an instance variable of this spout object.
    */
    void open(Map conf, TopologyContext context, SpoutOutputCollector collector);

    /**
    * Called when an ISpout is going to be shutdown. There is no guarentee that close
    * will be called, because the supervisor kill -9's worker processes on the cluster.
    *
    * <p>The one context where close is guaranteed to be called is a topology is
    * killed when running Storm in local mode.</p>
    */
    void close();

    /**
```

```
    * Called when a spout has been activated out of a deactivated mode.
    * nextTuple will be called on this spout soon. A spout can become activated
    * after having been deactivated when the topology is manipulated using the
    * `storm` client.
    */
    void activate();

    /**
    * Called when a spout has been deactivated. nextTuple will not be called while
    * a spout is deactivated. The spout may or may not be reactivated in the future.
    */
    void deactivate();

    /**
    * When this method is called, Storm is requesting that the Spout emit tuples to the
    * output collector. This method should be non-blocking, so if the Spout has no tuples
    * to emit, this method should return. nextTuple, ack, and fail are all called in a tight
    * loop in a single thread in the spout task. When there are no tuples to emit, it is courteous
    * to have nextTuple sleep for a short amount of time (like a single millisecond)
    * so as not to waste too much CPU.
    */
    void nextTuple();

    /**
    * Storm has determined that the tuple emitted by this spout with the msgId identifier
    * has been fully processed. Typically, an implementation of this method will take that
    * message off the queue and prevent it from being replayed.
    */
    void ack(Object msgId);

    /**
    * The tuple emitted by this spout with the msgId identifier has failed to be
    * fully processed. Typically, an implementation of this method will put that
    * message back on the queue to be replayed at a later time.
    */
    void fail(Object msgId);
}
```

源代码中已给出了较多的注释，这里不再赘述。关于nextTuple方法的实现，读者需要注意的是，由于nextTuple、ack和fail方法是在一个线程里面被调用的，如果nextTuple阻塞，其他方法也将被阻塞，这样会有许多意外情况发生，因此nextTuple必须是非阻塞的。任何的Spout都将利用nextTuple来发送信息。

ISpout的fail和ack回调方法仅仅给出了发送消息时所对应的MessageId，而没有给出具体的消息内容，这就意味着如果要实现消息的重传，用户则需要自己来维护那些已经发送的消息。

当Spout被设置为活跃或者不活跃时，会分别调用active回调方法和deactive回调方法将该状态通知给用户代码。这样当Spout处于非活跃状态时，nextTuple不会被调用。

在实现Spout的过程中，用户可以编写其构造函数，然而该构造函数并不会被实际调用。因为在提交Topology时，系统会调用Topology的构造函数（而非Spout的构造函数），并将产生的对象序列化成字符数组。每一个节点上的Spout对象都是通过反序列化得到的，这可能导致某些成

员没有被正确初始化。ISpout中的open回调函数会在对象被反序列后调用，我们应当在open方法中对对象的复杂成员进行初始化，而不应使用构造函数来完成这一过程。

3.7.2　IRichSpout

IRichSpout需要同时实现IComponent和ISpout接口，于是从含义上看它表示一个具有Spout功能的组件，其定义如下：

```
public interface IRichSpout extends ISpout, IComponent {

}
```

如果用户使用Java来构建Topology，IRichSpout以及下一节要介绍的IRichBolt是用来编写Topology组件的主要接口。

Storm是如何以及何时调用Spout内的各种方法呢？这将在第10章中详细解释。

3.8　Bolt 接口

Storm中定义的Bolt接口主要有IBolt、IRichBolt、IBasicBolt和IBatchBolt，它们的类关系如图3-3所示。

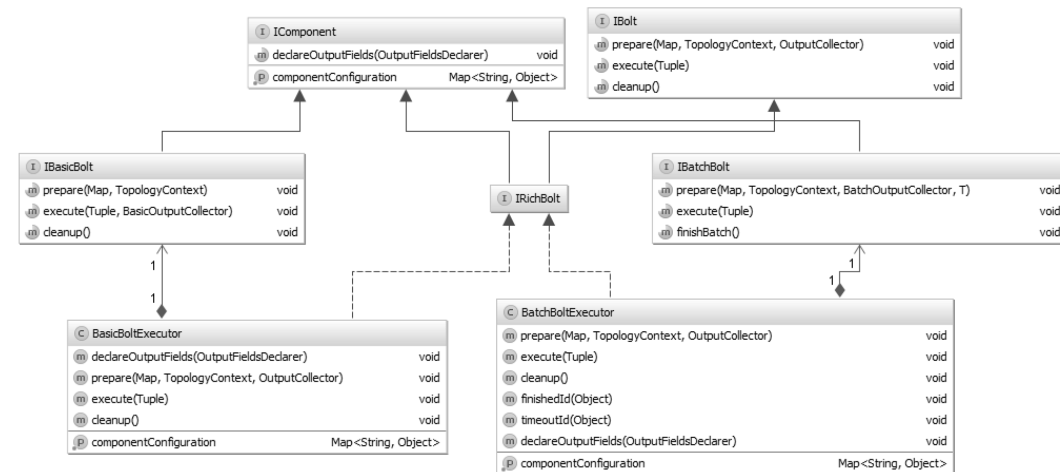


图3-3　Bolt接口的类关系图

3.8.1　IBolt

IBolt定义了Bolt的功能集合，其代码如下：

```
/**
* An IBolt represents a component that takes tuples as input and produces tuples
* as output. An IBolt can do everything from filtering to joining to functions
* to aggregations. It does not have to process a tuple immediately and may
* hold onto tuples to process later.
*
* <p>A bolt's lifecycle is as follows:</p>
*
* <p>IBolt object created on client machine. The IBolt is serialized into the topology
* (using Java serialization) and submitted to the master machine of the cluster (Nimbus).
* Nimbus then launches workers which deserialize the object, call prepare on it, and then
* start processing tuples.</p>
*
* <p>If you want to parameterize an IBolt, you should set the parameter's through its
* constructor and save the parameterization state as instance variables (which will
* then get serialized and shipped to every task executing this bolt across the cluster).</p>
*
* <p>When defining bolts in Java, you should use the IRichBolt interface which adds
* necessary methods for using the Java TopologyBuilder API.</p>
*/
public interface IBolt extends Serializable {
    /**
    * Called when a task for this component is initialized within a worker on the cluster.
    * It provides the bolt with the environment in which the bolt executes.
    *
    * <p>This includes the:</p>
    *
    * @param stormConf The Storm configuration for this bolt. This is the configuration provided to
       the topology merged in with cluster configuration on this machine.
    * @param context This object can be used to get information about this task's place within the
       topology, including the task id and component id of this task, input and output information,etc.
    * @param collector The collector is used to emit tuples from this bolt. Tuples can be emitted at
       any time, including the prepare and cleanup methods. The collector is thread-safe and should
       be saved as an instance variable of this bolt object.
    */
    void prepare(Map stormConf, TopologyContext context, OutputCollector collector);

    /**
    * Process a single tuple of input. The Tuple object contains metadata on it
    * about which component/stream/task it came from. The values of the Tuple can
    * be accessed using Tuple#getValue. The IBolt does not have to process the Tuple
    * immediately. It is perfectly fine to hang onto a tuple and process it later
    * (for instance, to do an aggregation or join).
    *
    * <p>Tuples should be emitted using the OutputCollector provided through the prepare method.
    * It is required that all input tuples are acked or failed at some point using the OutputCollector.
    * Otherwise, Storm will be unable to determine when tuples coming off the spouts
    * have been completed.</p>
    *
    * <p>For the common case of acking an input tuple at the end of the execute method,
    * see IBasicBolt which automates this.</p>
    *
    * @param input The input tuple to be processed.
    */
```

```
    void execute(Tuple input);

    /**
    * Called when an IBolt is going to be shutdown. There is no guarentee that cleanup
    * will be called, because the supervisor kill -9's worker processes on the cluster.
    *
    * <p>The one context where cleanup is guaranteed to be called is when a topology
    * is killed when running Storm in local mode.</p>
    */
    void cleanup();
}
```

Bolt是Storm中的基础运行单位，当其启动并有消息输入时，将调用execute方法来进行处理。与ISpout类似，IBolt对象在提交时也会被序列化为字节数组，具体的执行节点通过反序列化的方法得到该对象，并调用prepare回调方法。用户应将复杂对象的初始化放在prepare回调方法中实现，以保证每个具体对象都可以正确初始化。

对象被销毁时，将调用cleanup回调方法，但是Storm并不保证该方法一定被执行。

通常，在execute方法的实现中会对输入消息进行处理，这有可能产生新消息需要发送到下游节点，最后还要对输入的消息进行Ack操作。如果消息处理失败，则需对输入的消息进行Fail操作，这是保证Ack消息系统可以正常工作的基础。

3.8.2　IRichBolt

IRichBolt需要同时实现IComponent以及IBolt接口，其含义是一个具有Bolt功能的组件。在实际使用中，IRichBolt是实现Topology组件的主要接口，其定义如下：

```
public interface IRichBolt extends IBolt, IComponent {

}
```

3.8.3　IBasicBolt

IBasicBolt接口的定义与IBolt基本一致，具体的实现要求也与IBolt相同，它与IBolt的区别在于以下两点。

- 它的输出收集器使用的是BasicOutputCollector，并且该参数被放在了execute方法中而不是prepare中。
- 它实现了IComponent接口，这表明它可以用来定义Topology组件。

IBasicBolt的定义如下：

```
public interface IBasicBolt extends IComponent {
    void prepare(Map stormConf, TopologyContext context);
    /**
    * Process the input tuple and optionally emit new tuples based on the input tuple.
    *
    * All acking is managed for you. Throw a FailedException if you want to fail the tuple.
```

```
    */
    void execute(Tuple input, BasicOutputCollector collector);
    void cleanup();
}
```

这里解释一下为什么会有这个接口。IBasicBolt的主要作用是为用户提供一种更简单的Bolt编写方式。基于IBasicBolt编写的好处是Storm框架本身帮你处理了所发出消息的Ack、Fail和Anchor操作，这是由执行器BasicBoltExecutor实现的。

BasicBoltExecutor实现了IRichBolt接口，同时还包含了一个IBasciBolt成员变量用于调用的转发。它是基于装饰模式实现的，其定义如下：

```
1  public class BasicBoltExecutor implements IRichBolt {
2     public static Logger LOG = LoggerFactory.getLogger(BasicBoltExecutor.class);
3
4     private IBasicBolt _bolt;
5     private transient BasicOutputCollector _collector;
6
7     public BasicBoltExecutor(IBasicBolt bolt) {
8         _bolt = bolt;
9     }
10
11    public void declareOutputFields(OutputFieldsDeclarer declarer) {
12        _bolt.declareOutputFields(declarer);
13    }
14
15
16    public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {
17        _bolt.prepare(stormConf, context);
18        _collector = new BasicOutputCollector(collector);
19    }
20
21    public void execute(Tuple input) {
22        _collector.setContext(input);
23        try {
24            _bolt.execute(input, _collector);
25            _collector.getOutputter().ack(input);
26        } catch(FailedException e) {
27            if(e instanceof ReportedFailedException) {
28                _collector.reportError(e);
29            }
30            _collector.getOutputter().fail(input);
31        }
32    }
33
34    public void cleanup() {
35        _bolt.cleanup();
36    }
37
38    public Map<String, Object> getComponentConfiguration() {
39        return _bolt.getComponentConfiguration();
40    }
41 }
```

- ❑ 第4行定义了成员变量_bolt，它实现了IBasicBolt接口。
- ❑ 第5行定义了成员变量_collector，它实现了一个BasicOutputCollector对象，该对象提供了一些易用方法。例如，若发送消息时未指定输出流，它会将消息发送至id为default的流。
- ❑ 第11~13行实现了declareOutputFields方法，它实际上是在调用_bolt的declareOutputFields方法。
- ❑ 第16~19行调用_bolt的prepare方法，并实例化BasicOutputCollector。
- ❑ 第21~32行实现了execute方法。
- ❑ 第22行设置执行器运行的上下文，它表示经execute方法发送出去的消息都是由输入消息产生的，即输出的消息都将标记为输入消息所衍生出来的消息，这是使用IBasicBolt消息跟踪的重要一环。
- ❑ 第24行调用_bolt的execute方法。
- ❑ 第25行对输入的消息进行Ack操作。结合前面介绍的BasicOutputCollector以及第22行的设置可以知道，这一步意味着基于当前输入消息的处理及衍生消息的发送已经完成，此时就可以对该消息进行Ack操作了。
- ❑ 第26~31行中，Storm捕获所有的FailedException，并对输入的消息进行Fail操作。如果捕获的异常为ReportedFailedException实例，将调用reportError回调方法，给用户一个机会去处理异常。FailedException，是Storm定义的一种基本异常，用来进行消息的失败重发、事务的失败重试等操作，并不会导致Topology停止运行。

用户实现了IBasicBolt接口的Bolt对象以后，在构建Topology时，Storm会调用TopologyBuilder的setBolt方法设置该Bolt对象。SetBolt方法会用BasicBoltExecutor封装用户的实现类，这是Storm自动帮用户实现的，而且它还会调用可接收IRichBolt参数的重载方法完成Bolt设置。同时这也就解释了BasicBoltExecutor需要实现IRichBolt接口的原因。setBolt方法的代码如下：

```
public BoltDeclarer setBolt(String id, IBasicBolt bolt, Number parallelism_hint) {
    return setBolt(id, new BasicBoltExecutor(bolt), parallelism_hint);
}

public BoltDeclarer setBolt(String id, IRichBolt bolt, Number parallelism_hint) {
    validateUnusedId(id);
    initCommon(id, bolt, parallelism_hint);
    _bolts.put(id, bolt);
    return new BoltGetter(id);
}
```

3.8.4　IBatchBolt

区别于IBasicBolt接口，IBatchBolt主要用于Storm中的批处理。目前，Storm主要用该接口来实现可靠的消息传输，在这种情况下，批处理会比单一消息处理更为高效。Storm的事务Topology以及Trident主要是基于IBatchBolt的。相比前面的IBolt、IBasicBolt和IRichBolt，IBatchBolt中多了一个finishBatch方法，它在一个批处理结束时被调用。此外，IBatchBolt还去除了cleanup方法：

```
public interface IBatchBolt<T> extends Serializable, IComponent {
    void prepare(Map conf, TopologyContext context, BatchOutputCollector collector, T id);
    void execute(Tuple tuple);
    void finishBatch();
}
public abstract class BaseBatchBolt<T> extends BaseComponent implements IBatchBolt<T> { }
public abstract class BaseTransactionalBolt extends BaseBatchBolt<TransactionAttempt> {  }
```

IBatchBolt主要定义了3个方法，如下所示。

- prepare方法：用来初始化一个Batch。值得注意的是，在prepare方法中，最后一个参数是通用类型T，它可以用作该Batch的唯一标识。在IBatchBolt衍生的BaseTransactionalBolt中，T将被实例化为TransactionalAttempt。在目前的Storm实现中，每个事务都会对应一个Batch，而每个Batch的数据都会由一个新创建的IBatchBolt对象进行处理。于是在prepare方法中，需要传入一个用于标识batch的变量T。而在事务Topology中，Storm则利用TransactionAttempt作为标识。当一个Batch被成功处理之后，该Batch对应的IBatchBolt对象将被销毁，因此用户不能通过IBatchBolt对象自身保存需要在多个Batch间进行共享的数据。
- execute方法：用于处理属于该Batch的消息。
- finishBatch方法：该方法仅当这批消息被处理完时才会被调用。如果BatchBolt同时实现了ICommitter的接口，finishBatch方法只有当该Batch之前的所有Batch均被成功处理后才被调用。这既保证了强序关系，同时也是Storm中事务Topology的实现基础。至于如何保证finishBatch会在这批消息均被处理过后才调用，将在第15章中详细介绍。

跟IBasicBolt类似，使用IBatchBolt接口的用户不需要关心何时该对收到的消息进行Ack、Fail和Anchor操作，Storm框架内部通过BatchBoltExecutor自动帮我们实现了这些功能。

BatchBoltExecutor也实现了IRichBolt接口，它会为每个Batch创建与之相对应的BatchBolt对象，所有属于该Batch的消息都会使用对应的BatchBolt对象来处理。同时，它还实现了FinishedCallback和TimeoutCallback接口，这两个接口的作用我们会在后面给予介绍。BatchBoltExecutor的实现如下：

```
1 public class BatchBoltExecutor implements IRichBolt, FinishedCallback, TimeoutCallback {
2     public static Logger LOG = LoggerFactory.getLogger(BatchBoltExecutor.class);
3
4     byte[] _boltSer;
5     Map<Object, IBatchBolt> _openTransactions;
6     Map _conf;
7     TopologyContext _context;
8     BatchOutputCollectorImpl _collector;
9
10    public BatchBoltExecutor(IBatchBolt bolt) {
11        _boltSer = Utils.serialize(bolt);
12    }
13
14    @Override
15    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
16        _conf = conf;
17        _context = context;
18        _collector = new BatchOutputCollectorImpl(collector);
```

```
19          _openTransactions = new HashMap<Object, IBatchBolt>();
20      }
21
22      @Override
23      public void execute(Tuple input) {
24          Object id = input.getValue(0);
25          IBatchBolt bolt = getBatchBolt(id);
26          try {
27               bolt.execute(input);
28              _collector.ack(input);
29          } catch(FailedException e) {
30              LOG.error("Failed to process tuple in batch", e);
31              _collector.fail(input);
32          }
33      }
34
35      @Override
36      public void cleanup() {
37      }
38
39      @Override
40      public void finishedId(Object id) {
41          IBatchBolt bolt = getBatchBolt(id);
42          _openTransactions.remove(id);
43          bolt.finishBatch();
44      }
45
46      @Override
47      public void timeoutId(Object attempt) {
48          _openTransactions.remove(attempt);
49      }
50
51
52      @Override
53      public void declareOutputFields(OutputFieldsDeclarer declarer) {
54          newTransactionalBolt().declareOutputFields(declarer);
55      }
56
57      @Override
58      public Map<String, Object> getComponentConfiguration() {
59          return newTransactionalBolt().getComponentConfiguration();
60      }
61
62      private IBatchBolt getBatchBolt(Object id) {
63          IBatchBolt bolt = _openTransactions.get(id);
64          if(bolt==null) {
65              bolt = newTransactionalBolt();
66              bolt.prepare(_conf, _context, _collector, id);
67              _openTransactions.put(id, bolt);
68          }
69          return bolt;
70      }
71
72      private IBatchBolt newTransactionalBolt() {
```

```
73        return (IBatchBolt) Utils.deserialize(_boltSer);
74    }
75 }
```

- 第4行表示了内含BatchBolt对象的序列化字节数组。
- 第23~33行实现了execute方法，它规定输入消息的第1列用于标识Batch的id。在事务Topology中，输入消息的第1列为TransactionAttempt对象。第25行根据BatchId去获得一个BatchBolt对象。第28行对输入的消息进行Ack操作。第29~32行对捕获的FailedException进行处理，并对输入的消息进行Fail操作。
- 第40~44行实现了FinishedCallback接口，这里将调用finishBatch方法清理Batch对象。可以看到，BatchBolt对象在不同的Batch之间是不重复使用的。只要在属于某Batch的消息均被处理后，finishBatch方法才可以被调用。关于finishBatch调用时机，也将在第15章中进行分析。
- 第47~49行实现TimeoutCallback接口，它仅仅将缓存的BatchBolt对象删除，这对于清理不再使用的BatchBolt对象是很关键的。关于其调用时机，将在第15章进行分析。
- 第72~74行通过反序列化生成一个IBatchBolt对象。
- 第62~70行根据BatchId获得一个BatchBolt。类成员变量_openTransactions用来存储从每一个BatchId到BatchBolt的对应关系。如果BatchId对应的BatchBolt并不存在，getBatchBolt方法将调用newTransactionalBolt生成一个。第66行调用新创建的BatchBolt的prepare方法。

理解IBatchBolt的生命周期是很关键的。IBatchBolt在系统收到属于某Batch的第一条消息时被创建，而在所有的消息都处理完成之后再被销毁。Storm中采用反序列化对象的方式来弥补不断创建IBatchBolt对象所带来的负担。

3.8.5 小结

这一节主要介绍了Storm中几种基本的Bolt接口，这里简单总结一下。

- IRichBolt：Storm中最常用来定义Topology组件的接口。它十分灵活，用户可以通过其实现各种控制逻辑，并且能控制何时进行Ack、Fail和Anchor操作。
- IBasicBolt：Storm中提供的定义简单逻辑的Topology组件接口。对于这种Bolt，Storm内置实现了Ack、Fail和Anchor的机制，用户基于它实现自己的Bolt也比较简单。但是它的使用是有限制的，基于收到的某条消息衍生出来的所有消息必须在一次execute中发送出去（或者需要对消息进行缓存并且编号，参考第27章），否则内置的Ack机制将不能保证Bolt的正常工作。所以，用户应该避免使用该类型的Bolt来做诸如聚集或者连接的操作。
- IBatchBolt：它是Storm提供的用来处理批量数据的接口。目前，它只用于事务Topology中，它是Storm实现事务Topology的基础。

3.9　Storm 数据结构

storm.thrift文件定义了Storm中用到的最底层的数据结构。Thrift是Apache下面的跨语言框架，它可以基于Thrift定义文件产生不同语言的代码。关于Thrift的详细内容，请访问http://thrift.apache.org/进行学习。下面我们来看一下Storm中使用的Thrift数据结构的定义。

3.9.1　GlobalStreamId

GlobalStreamId结构的定义如下所示：

```
struct GlobalStreamId {
    1: required string componentId;
    2: required string streamId;
    #Going to need to add an enum for the stream type (NORMAL or FAILURE)
}
```

在Storm中，流（stream）是一个非常重要的概念。流可以理解为消息的渠道，每种类型的消息可以用一个流来表示。组件可以向多个流发送消息，也可以从多个流接收消息。GlobalStreamId用来标记每一个组件的流信息，它含有两个域，一个为componentId，表示该流属于哪个组件；另外一个是streamId，它是流的标识。不同的组件之间可以使用相同的streamId，例如在默认情况下，用户接收和发送的数据均来自于streamId为default的流，只不过它们隶属于不同的组件。

3.9.2　消息分组方式

分组方式决定了组件所发送的消息将以何种方式到达接收端，这是Storm具有可扩展性的基础。例如，使用随机分组，节点可以发送一组消息到下游的多个节点，每个节点收到这组消息的一部分，而且这些节点可以分布在不同的机器上，从而达到了并行处理的目的。Grouping的定义如下所示：

```
union Grouping {
    1: list<string> fields; //empty list means global grouping
    2: NullStruct shuffle; // tuple is sent to random task
    3: NullStruct all; // tuple is sent to every task
    4: NullStruct none; // tuple is sent to a single task (storm's choice) -> allows storm to optimize
        the topology by bundling tasks into a single process
    5: NullStruct direct; // this bolt expects the source bolt to send tuples directly to it
    6: JavaObject custom_object;
    7: binary custom_serialized;
    8: NullStruct local_or_shuffle; // prefer sending to tasks in the same worker process, otherwise
        shuffle
}
```

Grouping被定义为union类型，即表示节点之间只能采取一种分组方式。

3.9.3 StreamInfo

StreamInfo含有两个数据成员，一个表示输出的字段名列表，另一个则表示是否为直接流。目前，Storm限制只有直接流才能采用直接分组方式。StreamInfo的定义如下：

```
struct StreamInfo {
    1: required list<string> output_fields;
    2: required bool direct;
}
```

3.9.4 ShellComponent

ShellComponent结构用于与非Java语言的交互，目前通过标准输入输出来交换数据：

```
struct ShellComponent {
    // should change this to 1: required list<string> execution_command;
    1: string execution_command;
    2: string script;
}
```

3.9.5 ComponentObject

ComponentObject是一个联合体，它或者为串行化后的Java对象，或者为ShellComponent对象，抑或为Java对象。通常，第一种和第三种更为常见。该联合体的定义如下：

```
union ComponentObject {
    1: binary serialized_java;
    2: ShellComponent shell;
    3: JavaObject java_object;
}
```

3.9.6 ComponentCommon

ComponentCommon是用来表示Topology的基础对象，通过使用TopologyBuilder对象可更加方便地创建这个数据结构。后面我们会进一步讨论TopologyBuilder。ComponentCommon的定义如下：

```
struct ComponentCommon {
    1: required map<GlobalStreamId, Grouping> inputs;
    2: required map<string, StreamInfo> streams; //key is stream id
    3: optional i32 parallelism_hint; //how many threads across the cluster should be dedicated to this
         component

    // component specific configuration respects:
    // topology.debug: false
    // topology.max.task.parallelism: null // can replace isDistributed with this
    // topology.max.spout.pending: null
```

```
    // topology.kryo.register // this is the only additive one

    // component specific configuration
    4: optional string json_conf;
}
```

- inputs：表示该组件将从哪些GlobalStreamId以何种分组方式接收数据，其中GlobalStreamId即为某个组件上面定义的一个流。
- streams：表示该组件要输出的所有流。它给定了streamId以及StreamInfo。在StreamInfo中，我们定义了每个输出流的字段名列表，以及该流的消息分组是否为直接分组方式。
- parallelism_hint：表示组件的并行度，即有多少个线程。要注意，这些线程可能分布在不同的机器以及进程空间中，其默认值为1。
- json_conf：保存与该组件相关的设置，它可设置的参数如表3-1所示。

表3-1　组件的配置项

参数名称	含　义
topology.debug	设置为true时，Storm会打印出所有发送出去的消息，包括系统消息，例如Ack消息
topology.max.task.parallelism	任务的最大并行度，通常用于测试
topology.max.spout.pending	仅仅作用于每个Spout节点，表示最多能有多少没被Ack或Fail的消息在系统中运行。若超出此限制，Spout将转入非活跃状态，即不会有新的消息发送出来。 只有发送的消息指定了messageId时，该参数才会生效。messageId也是Ack系统正常工作所必需的
topology.kryo.register	kryo序列化的注册列表。它是Storm底层的序列化框架 （https://github.com/EsotericSoftware/kryo） 该列表的内容可以是类的名称，kryo会根据该类名称自动序列化所有对象的成员变量，也可以是一个实现了com.esotericsoftware.kryo.Serializer的类名称

3.9.7　SpoutSpec

SpoutSpec包含两个成员，一个为实现具体Spout逻辑的spout_object对象，另外一个是用来描述其输入输出的common对象。有时可以将Spout设置为单点的，如事务Topology中的协调Spout节点。可以通过设置topology.max.task.parallelism参数进行控制。SpoutSpec的定义如下所示：

```
struct SpoutSpec {
    1: required ComponentObject spout_object;
    2: required ComponentCommon common;
    // can force a spout to be non-distributed by overriding the component configuration
    // and setting TOPOLOGY_MAX_TASK_PARALLELISM to 1
}
```

3.9.8　Bolt

Bolt的数据结构与SpoutSpec是一致的，其中包含两个成员，bolt_object是包含具体逻辑的

对象，common则描述了输入输出：

```
struct Bolt {
    1: required ComponentObject bolt_object;
    2: required ComponentCommon common;
}
```


3.9.9　StormTopology

StormTopolgy用于描述Topology的组成，它包含了一些SpoutSpec和Bolt，每个SpoutSpec或Bolt都会有一个全局唯一的名字。目前，state_spouts基本上没有用到。

```
struct StormTopology {
    //ids must be unique across maps
    // #workers to use is in conf
    1: required map<string, SpoutSpec> spouts;
    2: required map<string, Bolt> bolts;
    3: required map<string, StateSpoutSpec> state_spouts;
}
```

3.9.10　TopologySummary

TopologySummary描述了由用户提交的Topology的基本情况，例如Topology分布在多少个Worker上面，使用了多少个Executor，以及一共有多少个Task等。这个数据结构主要供Nimbus使用，以返回UI请求的数据，其定义如下：

```
struct TopologySummary {
    1: required string id;
    2: required string name;
    3: required i32 num_tasks;
    4: required i32 num_executors;
    5: required i32 num_workers;
    6: required i32 uptime_secs;
    7: required string status;
}
```

3.9.11　SupervisorSummary

SupervisorSummary描述了每一个Supervisor的基本信息。这里Supervisor代表了机器，一个Supervisor上可以启动多个Worker（通常为4个），每个Worker上面又可以启动多个Executor。这个数据结构也是在Nimbus返回UI请求的数据时会被用到，其中num_used_workers表示已经占用的Worker数目，其定义如下：

```
struct SupervisorSummary {
    1: required string host;
```

```
    2: required i32 uptime_secs;
    3: required i32 num_workers;
    4: required i32 num_used_workers;
    5: required string supervisor_id;
}
```

3.9.12　ClusterSummary

ClusterSummary保存了集群中所包含的Supervisor的数目及其基本信息，还保存了正在集群上运行的Topology的基本信息。Nimbus在返回Storm UI请求的数据时也会用到该数据结构。其定义如下：

```
struct ClusterSummary {
    1: required list<SupervisorSummary> supervisors;
    2: required i32 nimbus_uptime_secs;
    3: required list<TopologySummary> topologies;
}
```

3.9.13　BoltStats

BoltStats结构用于Bolt的运行统计，其定义如下：

```
struct BoltStats {
    1: required map<string, map<GlobalStreamId, i64>> acked;
    2: required map<string, map<GlobalStreamId, i64>> failed;
    3: required map<string, map<GlobalStreamId, double>> process_ms_avg;
    4: required map<string, map<GlobalStreamId, i64>> executed;
    5: required map<string, map<GlobalStreamId, double>> execute_ms_avg;
}
```

3.9.14　SpoutStats

SpoutStats用于Spout的运行统计，其定义如下：

```
struct SpoutStats {
  1: required map<string, map<string, i64>> acked;
  2: required map<string, map<string, i64>> failed;
  3: required map<string, map<string, double>> complete_ms_avg;
}
```

3.9.15　统计信息

下面的数据结构包含了在Storm UI上显示的数据，例如每个Executor发送及传输消息的数目等：

```
union ExecutorSpecificStats {
    1: BoltStats bolt;
    2: SpoutStats spout;
```

```
}

// Stats are a map from the time window (all time or a number indicating number of seconds in the window)
// to the stats. Usually stats are a stream id to a count or average.
struct ExecutorStats {
    1: required map<string, map<string, i64>> emitted;
    2: required map<string, map<string, i64>> transferred;
    3: required ExecutorSpecificStats specific;
}

struct ExecutorInfo {
    1: required i32 task_start;
    2: required i32 task_end;
}

struct ExecutorSummary {
    1: required ExecutorInfo executor_info;
    2: required string component_id;
    3: required string host;
    4: required i32 port;
    5: required i32 uptime_secs;
    7: optional ExecutorStats stats;
}

struct TopologyInfo {
    1: required string id;
    2: required string name;
    3: required i32 uptime_secs;
    4: required list<ExecutorSummary> executors;
    5: required string status;
    6: required map<string, list<ErrorInfo>> errors;
}
```

3.9.16 DRPC

DRPCRequest定义了DRPC请求的数据结构，它包含函数的参数列表func_args以及请求的request_id。由于在获取请求时需要传入函数名，所以DRPCRequest对象中并不包含函数名。

DRPCExecutionException是异常类型，它在处理请求失败或者超时时会用到，它会采用不同的出错消息msg对不同异常状况进行区分。

DistributedRPC和Distribute RPCInvocations均为服务类型，它们描述了DRPC服务器提供的服务接口，可以用于产生服务器的访问代码。它们的区别在于前者负责处理用户请求，后者则供DRPC Spout获取请求或供Bolt向DRPC服务器发送结果。DRPCRequest的定义如下：

```
struct DRPCRequest {
    1: required string func_args;
    2: required string request_id;
}

exception DRPCExecutionException {
```

```
    1: required string msg;
}

service DistributedRPC {
    string execute(1: string functionName, 2: string funcArgs) throws (1: DRPCExecutionException e);
}

service DistributedRPCInvocations {
    void result(1: string id, 2: string result);
    DRPCRequest fetchRequest(1: string functionName);
    void failRequest(1: string id);
}
```

3.10　基本 Topology 构建器

这一节主要介绍TopologyBuilder以及它所提供的配置声明和组件声明的实现类,最后结合一个例子来解释它的用法。

3.10.1　TopologyBuilder

TopologyBuilder是一个工具类，用于构建Topology。在Storm中，Topology实际上是Thrift的数据结构,其过于描述化的特性并不便于我们使用,而TopologyBuilder提供了更为方便的构建方法。它的主要方法有setSpout、setBolt以及它们的重载方法。其最终目的是创建前面介绍过的StormTopology对象。仔细阅读代码中的注释对理解代码是很有帮助的。TopologyBuilder类的定义如下：

```
1 public class TopologyBuilder {
2     private Map<String, IRichBolt> _bolts = new HashMap<String, IRichBolt>();
3     private Map<String, IRichSpout> _spouts = new HashMap<String, IRichSpout>();
4     private Map<String, ComponentCommon> _commons = new HashMap<String, ComponentCommon>();
5     private Map<String, StateSpoutSpec> _stateSpouts = new HashMap<String, StateSpoutSpec>();
6
7     public StormTopology createTopology() {
8         Map<String, Bolt> boltSpecs = new HashMap<String, Bolt>();
9         Map<String, SpoutSpec> spoutSpecs = new HashMap<String, SpoutSpec>();
10        for(String boltId: _bolts.keySet()) {
11            IRichBolt bolt = _bolts.get(boltId);
12            ComponentCommon common = getComponentCommon(boltId, bolt);
13                boltSpecs.put(boltId, new Bolt(ComponentObject.serialized_java(Utils.serialize
                      (bolt)),common));
14        }
15        for(String spoutId: _spouts.keySet()) {
16            IRichSpout spout = _spouts.get(spoutId);
17            ComponentCommon common = getComponentCommon(spoutId, spout);
18            spoutSpecs.put(spoutId, new SpoutSpec(ComponentObject.serialized_java(Utils.
                 serialize(spout)), common));
19
20        }
21        return new StormTopology(spoutSpecs,
```

```
22              boltSpecs,
23              new HashMap<String, StateSpoutSpec>());
24     }
25
26 public BoltDeclarer setBolt(String id, IRichBolt bolt) {
27        return setBolt(id, bolt, null);
28     }
29
30     /**
31      * Define a new bolt in this topology with the specified amount of parallelism.
32      *
33      * @param id the id of this component. This id is referenced by other components that want to
          consume this bolt's outputs.
34      * @param bolt the bolt
35      * @param parallelism_hint the number of tasks that should be assigned to execute this bolt.
          Each task will run on a thread in a process somewhere around the cluster.
36      * @return use the returned object to declare the inputs to this component
37      */
38 public BoltDeclarer setBolt(String id, IRichBolt bolt, Number parallelism_hint) {
39      validateUnusedId(id);
40      initCommon(id, bolt, parallelism_hint);
41      _bolts.put(id, bolt);
42      return new BoltGetter(id);
43    }
44
45    public BoltDeclarer setBolt(String id, IBasicBolt bolt) {
46        return setBolt(id, bolt, null);
47    }
48
49    public BoltDeclarer setBolt(String id, IBasicBolt bolt, Number parallelism_hint) {
50        return setBolt(id, new BasicBoltExecutor(bolt), parallelism_hint);
51    }
52
53    /**
54      * Define a new spout in this topology.
55      *
56      * @param id the id of this component. This id is referenced by other components that want
           to consume this spout's outputs.
57      * @param spout the spout
58      */
59    public SpoutDeclarer setSpout(String id, IRichSpout spout) {
60       return setSpout(id, spout, null);
61    }
62
63    /**
64      * Define a new spout in this topology with the specified parallelism. If the spout declares
65      * itself as non-distributed, the parallelism_hint will be ignored and only one task
66      * will be allocated to this component.
67      *
68      * @param id the id of this component. This id is referenced by other components that want
           to consume this spout's outputs.
69      * @param parallelism_hint the number of tasks that should be assigned to execute this
           spout. Each task will run on a thread in a process somwehere around the cluster.
70      * @param spout the spout
```

```
71    */
72   public SpoutDeclarer setSpout(String id, IRichSpout spout, Number parallelism_hint) {
73       validateUnusedId(id);
74       initCommon(id, spout, parallelism_hint);
75       _spouts.put(id, spout);
76       return new SpoutGetter(id);
77   }
78
79 private void validateUnusedId(String id) {
80     if(_bolts.containsKey(id)) {
81         throw new IllegalArgumentException("Bolt has already been declared for id " + id);
82     }
83     if(_spouts.containsKey(id)) {
84         throw new IllegalArgumentException("Spout has already been declared for id " + id);
85     }
86     if(_stateSpouts.containsKey(id)) {
87          throw new IllegalArgumentException("State spout has already been declared for id"+id);
88     }
89 }
90
91 private ComponentCommon getComponentCommon(String id, IComponent component) {
92     ComponentCommon ret = new ComponentCommon(_commons.get(id));
93
94     OutputFieldsGetter getter = new OutputFieldsGetter();
95     component.declareOutputFields(getter);
96     ret.set_streams(getter.getFieldsDeclaration());
97     return ret;
98 }
99
100 private void initCommon(String id, IComponent component, Number parallelism) {
101    ComponentCommon common = new ComponentCommon();
102    common.set_inputs(new HashMap<GlobalStreamId, Grouping>());
103    if(parallelism!=null) common.set_parallelism_hint(parallelism.intValue());
104    Map conf = component.getComponentConfiguration();
105    if(conf!=null) common.set_json_conf(JSONValue.toJSONString(conf));
106    _commons.put(id, common);
107   }
108 }
```

- 第2~5行定义了类成员变量。_bolts包含了所有的Bolt对象，它们均为IRichBolt类型。_spout包含了所有的Spout对象，均为IRichSpout类型。_commons则包含了所有的Bolt及Spout对象。_stateSpouts包含了StateSpout对象，StateSpout是具有同步功能的Spout对象，不过可能由于其还处于试验阶段，用到的地方并不多。
- 第7~24行根据输入的Bolt和Spout对象构建StormTopology对象。从第13行以及第18行可以看出，在StormTopology中Spout和Bolts均为对象序列化过后得到的字节数组。
- 第26~51行定义setBolt方法及其各种重载方法。从第50行代码可以看到，setBolt方法会利用BasicBoltExecutor包装输入的IBasicBolt对象，其中BasicBoltExecutor还实现了消息的跟踪及发送。第39行会检测输入的组件ID当前是否是唯一的。第40~41行用于生成ComponentCommon对象。第42行返回一个BoltGetter对象，根据上节的分析，将利用其为

Bolt对象添加输入。

- 第72~77行定义了setSpout方法，它类似于setBolt方法，也将产生ComponentCommon对象。
- 第91~98行定义了getComponentCommon方法，该方法主要定义输出的流。
- 第100~107行定义了initCommon方法，它主要对ComponentCommon对象进行初始化，例如设置并行度和标准配置等。

3.10.2 ConfigGetter

ConfigGetter是定义在TopologyBuilder.java中的一个类，它实现了ComponentConfigurationDeclarer接口，并且继承自BaseConfigurationDeclarer类。其定义如下所示：

```
protected class ConfigGetter<T extends ComponentConfigurationDeclarer> extends BaseConfiguration
    Declarer<T> {
    String _id;

    public ConfigGetter(String id) {
        _id = id;
    }

    @Override
    public T addConfigurations(Map conf) {
        if(conf!=null && conf.containsKey(Config.TOPOLOGY_KRYO_REGISTER)) {
            throw new IllegalArgumentException("Cannot set serializations for a component using fluent
                API");
        }
        String currConf = _commons.get(_id).get_json_conf();
        _commons.get(_id).set_json_conf(mergeIntoJson(parseJson(currConf), conf));
        return (T) this;
    }
}
```


其中_id代表组件的唯一标识符。在通常的非事务流处理中，不能设定组件的序列化方法，只能采用系统默认的序列化方法。ConfigGetter会根据新设置的配置项覆盖组件默认的配置项。具体的实现过程中，最终的配置项会被序列化为JSON格式。

3.10.3 SpoutGetter和BoltGetter

SpoutGetter基本上实现了与ConfigGetter相同的功能，唯一的区别是Spout不需要对输入进行声明，其定义如下：

```
protected class SpoutGetter extends ConfigGetter<SpoutDeclarer> implements SpoutDeclarer {
    public SpoutGetter(String id) {
        super(id);
    }
}
```

BoltGetter不但继承ConfigGetter类实现了对于Bolt 组件的配置定制，同时还实现了接口

BoltDeclarer。由于其方法较多且极为类似，这里仅以fieldsGrouping为例进行分析。BoltGetter类的代码如下：

```
1  protected class BoltGetter extends ConfigGetter<BoltDeclarer> implements BoltDeclarer {
2      private String _boltId;
3
4      public BoltGetter(String boltId) {
5          super(boltId);
6          _boltId = boltId;
7      }
8
9      public BoltDeclarer fieldsGrouping(String componentId, Fields fields) {
10         return fieldsGrouping(componentId, Utils.DEFAULT_STREAM_ID, fields);
11      }
12
13      public BoltDeclarer fieldsGrouping(String componentId, String streamId, Fields fields) {
14          return grouping(componentId, streamId, Grouping.fields(fields.toList()));
15      }
16      private BoltDeclarer grouping(String componentId, String streamId, Grouping grouping) {
17         _commons.get(_boltId).put_to_inputs(new GlobalStreamId(componentId, streamId),
           grouping);
18         return this;
19      }
20
21      @Override
22      public BoltDeclarer grouping(GlobalStreamId id, Grouping grouping) {
23          return grouping(id.get_componentId(), id.get_streamId(), grouping);
24      }
25  }
```

- 第2行定义了_boltld变量，用于保存Bolt组件的唯一标识符。
- 第4~7行为构造函数。调用ConfigGetter的构造函数，可以实现配置项的定制功能。
- 第9~15行定义了fieldsGrouping及其重载方法，其参数包括componentId、streamId以及fields（用来分组的字段名列表）。如果没有指定streamId，默认将使用default作为streamId。
- 第16~19行是InputDeclarer接口最主要的实现方法。它的输入含义为：按照grouping参数定义的分组方式，把参数componentId指定的组件中以参数streamId定义的流放在本组件的输入定义中。第18行代码返回this指针，这样可以通过函数级联的方式为本组件添加更多的输入。
- 第22~24行是一个重要的重载，其输入为GlobalStreamId以及分组方式。

3.10.4　一个简单例子

这里举一个简单的例子，示例代码如下：

```
1 TopologyBuilder builder = new TopologyBuilder();
2     builder.setSpout("spout1", new Spout1(), 1);
```

```
3    builder.setSpout("spout2", new Spout2(), 2);
4    builder.setBolt("bolt, new Bolt1(), 1)
5      .shuffleGrouping("spout1");
6      .fieldsGrouping("spout2", new Fields( “F” ));
```

- 第1行代码定义了TopologyBuilder对象，这个对象已在3.10.1节介绍。
- 第2~3行代码设置了两个Spout。
- 第4行代码设置了一个Bolt，setBolt方法将返回一个BoltDeclarer对象。接着第5~6行代码将为该Bolt添加两个输入，一个以shuffleGrouping的方式连到spout1，另外一个则以fieldsGrouping的方式连到spout2。

3.11 异常处理

在Storm中，所有的FailedException都是被捕获的，它用以表示消息处理失败。在捕获该异常之后，Storm将对消息进行Fail操作，并通过Ack系统将消息处理失败的情况反映到Spout端，然后Spout端会根据用户定义的逻辑重传或者忽略消息等。

用户代码可以通过抛出FailedException来通知Storm对失败的消息进行处理，这可能会导致消息重传。若是由于用户代码的缺陷导致消息处理失败，则不应抛出该异常，因为即便重传也无济于解决问题。更好的处理方式是：或者尽快结束Topology的运行以解决问题，或者干脆直接忽略掉该消息。FailedException类的定义如下所示：

```
public class FailedException extends RuntimeException {
    public FailedException() {
        super();
    }

    public FailedException(String msg) {
        super(msg);
    }

    public FailedException(String msg, Throwable cause) {
        super(msg, cause);
    }

    public FailedException(Throwable cause) {
        super(cause);
    }
}
```

可以看到，FailedException继承自RuntimeException。根据Java的语言规范，继承自RuntimeException的异常类不需要放置于使用类的函数声明中。

第4章 基础函数和工具类

本章将介绍Storm源代码中那些通用且关键的基础函数和工具类，很多Storm组件的实现都依赖于它们。读者可以进行简单的阅读来了解其大概，也可以先跳过这一章，待用到这里的函数或工具类时再回来查阅。

4.1 计时器

Storm使用计时器线程来处理一些周期性调度事件。与计时器相关的操作主要包括创建计时器线程、查看线程是否活跃、向线程中加入新的待调度事件，以及取消计时器线程等，下面简要介绍这些操作。

4.1.1 mk-timer

mk-timer函数用于创建一个计时器线程。其基本思想为：首先定义一个优先级队列，队列中的元素类型为<目标执行时间，执行函数，序列号>，然后在当前时间大于队列中的目标执行时间时，取出队列中的元素并调用其执行函数。mk-timer的代码如下：

```
1  (defnk mk-timer [:kill-fn (fn [& _] )]
2      (let [queue (PriorityQueue. 10
3           (reify Comparator
4               (compare [this o1 o2]
5                   (- (first o1) (first o2))
6                   )
7               (equals [this obj]
8                   true
9                   )))
10      active (atom true)
11      lock (Object.)
12      notifier (Semaphore. 0)
13      timer-thread (Thread.
14          (fn []
15              (while @active
16                  (try
17                  (let [[time-secs _ _ :as elem] (locking lock (.peek queue))]
18                      (if (and elem (>= (current-time-secs) time-secs))
```

```
19                          ;; imperative to not run the function inside the timer lock
20                          ;; otherwise, it's possible to deadlock if function deals with
                                 other locks
21                          ;; (like the submit lock)
22                          (let [afn (locking lock (second (.poll queue)))]
23                          (afn))
24                          (Time/sleep 1000)
25                          ))
26                  (catch Throwable t
27                           ;; because the interrupted exception can be wrapped in a runtimeexception
28                           (when-not (exception-cause? InterruptedException t)
29                           (kill-fn t)
30                           (reset! active false)
31                           (throw t))
32                           )))
33        (.release notifier)))]
34    (.setDaemon timer-thread true)
35    (.setPriority timer-thread Thread/MAX_PRIORITY)
36    (.start timer-thread)
37    {:timer-thread timer-thread
38    :queue queue
39    :active active
40    :lock lock
41    :cancel-notifier notifier}))
```


- 第2~9行定义函数变量queue为优先级队列PriorityQueue，其初始化大小为10；同时传入一个比较器Comparator，并按照元素的第一部分（即目标执行时间）进行比较，将队列中的元素按照执行时间由小到大排列。
- 第10行的active变量用于表明该定时器是否处于活跃状态。
- 第11行的lock为Object对象，用来为队列对象的操作加锁。这是由于向队列中插入元素以及处理队列中元素这两项操作是在不同的线程里执行的。
- 第12行的notifier为信号量。计时器线程在结束时，会调用该信号量的release方法释放。
- 第13~33行定义计时器线程。线程的执行函数为一个while循环体。
  - 第17行调用队列的peek函数获得队列的一个元素。:as操作将元素的所有项放入变量elem中，即elem的内容为<目标执行时间，执行函数，元素序号>。
  - 在第18行中，若elem不为null，并且当前时间大于或等于元素的执行时间time-secs，则调用poll方法获取该元素，并调用second方法获得元素的第二列，即待执行函数afn，然后调用待执行函数afn。若elem不为null，线程则睡眠1秒。
  - 第26~32行对异常进行处理。若不为InterruptedException异常，则调用kill-fn，设置active为false，并最终抛出异常。kill-fn为构建定时器时传入的参数，用于对异常情况进行处理，该函数的参数为抛出的异常。例如，在Worker的mk-halting-timer函数中，kill-fn会记录异常，然后结束进程。该函数的代码如下：

```
(defn mk-halting-timer []
   (mk-timer :kill-fn (fn [t]
```

```
            (log-error t "Error when processing event")
            (halt-process! 20 "Error when processing an event")
            )))
```

- 在第33行中，当线程的循环结束后，调用notifier的release方法。
- 第34~36行将线程设置为后台线程并赋予其最高优先级，然后启动线程。
- 第37~41行设置返回值。这里返回的是一个哈希表，它包含了定义的计时器线程、优先级队列、活跃状态和锁对象等信息。返回后，外部环境便可以使用这些信息。

4.1.2 check-active!

check-active! 函数用于检测定时器的active变量，若定时器处于非活跃状态，则抛出异常，其代码如下：

```
(defn- check-active! [timer]
    (when-not @(:active timer)
        (throw (IllegalStateException. "Timer is not active"))))
```

4.1.3 schedule

schedule函数用于将一个事件注册到定时器中，其参数包括定时器timer、延迟执行的时间delay-secs、事件的执行函数afn，以及确定是否检测定时器活跃的参数check-active（其默认值为true）。该函数的代码如下：

```
1 (defnk schedule [timer delay-secs afn :check-active true]
2    (when check-active (check-active! timer))
3    (let [id (uuid)
4       ^PriorityQueue queue (:queue timer)]
5       (locking (:lock timer)
6          (.add queue [(+ (current-time-secs) delay-secs) afn id])
7          )))
```

第3行调用uuid函数产生一个随机数来代表该事件的事件号。第4~6行获取定时器的队列对象，并向其中添加元素。事件的目标执行时间为当前时间加上delay-secs。

4.1.4 schedule-recurring

schedule函数只执行一次注册函数，而schedule-recurring则可以按照设定的时间反复执行，其基本思路为执行完当前函数后调用函数schedule再次注册自身。其代码如下：

```
(defn schedule-recurring [timer delay-secs recur-secs afn]
    (schedule timer
        delay-secs
         (fn this []
            (afn)
            (schedule timer recur-secs this :check-active false)) ; this avoids a race condition with
```

```
                cancel-timer
        ))
```

delay-secs为第一次的延迟执行时间，而recur-secs则为事件循环执行的间隔时间。重新注册函数时将不会对定时器的活跃状态进行检查。否则，同时调用cancel-timer和schedule函数可能会有竞争，而先调用cancel-timer然后调用schedule方法，则check-active函数又会抛出异常，这都是不希望看到的。

4.1.5 cancel-timer

cancel-timer函数用于结束一个定时器，它会调用定时器notifier的acquire操作，等待定时器正常结束。其代码如下：

```
(defn cancel-timer [timer]
    (check-active! timer)
    (locking (:lock timer)
        (reset! (:active timer) false)
        (.interrupt (:timer-thread timer)))
    (.acquire (:cancel-notifier timer)))
```

4.2 async-loop

async-loop函数是Storm中一个非常重要的工具函数，Worker和Executor中的许多线程都是通过该函数来启动的。该函数使用一个线程来循环调用传入的函数afn，同时要求被调用的afn在执行结束后返回一个时间间隔，并将其作为与下次调用之间需等待的时间间隔。该函数的代码如下：

```
1 (defnk async-loop [afn
2     :daemon false
3     :kill-fn (fn [error] (halt-process! 1 "Async loop died!"))
4     :priority Thread/NORM_PRIORITY
5     :factory? false
6     :start true]
7     (let [thread (Thread.
8          (fn []
9              (try-cause
10                (let [afn (if factory? (afn) afn)]
11                     (loop []
12                         (let [sleep-time (afn)]
13                            (when-not (nil? sleep-time)
14                               (sleep-secs sleep-time)
15                               (recur))
16                            )))
17                (catch InterruptedException e
18                     (log-message "Async loop interrupted!")
19                     )
20                (catch Throwable t
21                     (log-error t "Async loop died!")
```

```
22                      (kill-fn t)
23                     ))
24                 ))]
25     (.setDaemon thread daemon)
26     (.setPriority thread priority)
27     (when start
28         (.start thread))
29     ;; should return object that supports stop, interrupt, join, and waiting?
30     (reify SmartThread
31        (start [this]
32             (.start thread))
33        (join [this]
34             (.join thread))
35        (interrupt [this]
36             (.interrupt thread))
37        (sleeping? [this]
38             (Time/isThreadWaiting thread)
39             ))
40        ))
```

- 第7~24行定义了一个线程来执行前面计算的afn函数。在第10行中，若factory? 为true，则代表输入的函数afn为一个工厂方法，此时需要调用afn函数来产生目标调用函数并赋值给afn。例如，Executor中的mk-threads方法在调用async-loop时传入的即为工厂函数，该函数被执行后将返回Executor的消息循环函数。
  - 第12~16行调用afn函数。afn函数返回一个等待时间，若该时间为null，则循环结束。
  - 第17~22行对异常情况进行处。若为InterruptedException，则只是打印消息并退出循环体，其他情况下则调用kill-fn函数（它是在调用async-loop函数时被传入的）。
- 第25~28行设置参数并启动线程。
- 第30~39行实例化SmartThread对象，用于对async-loop中的线程对象进行操作。

4.3　event-manager

event-manager函数会创建一个单独的线程来处理事件，创建时可以指定该线程是否是一个后台线程，处理过程中发生的任何错误都会导致该线程退出。该函数创建了一个LinkedBlockingQueue用来存储要执行的事件，同时还定义了3个变量added、processed和running。added用来记录当前总共有多少要执行的事件，processed记录了当前已经处理了多少事件，running则用于标识该线程是否活跃。

event-manager函数的处理逻辑很简单，就是一直不断地尝试从LinkedBlockingQueue中获取事件，获取到了就处理并更新processed的值，如果没有新事件，该获取方法会被阻塞直到有新的事件加入到队列中。目前，该函数主要用在Supervisor中，其定义如下：

```
1 (defprotocol EventManager
2     (add [this event-fn])
3     (waiting? [this])
4   (shutdown [this]))
```

```
5
6 (defn event-manager
7     "Creates a thread to respond to events. Any error will cause process to halt"
8     [daemon?]
9     (let [added (atom 0)
10        processed (atom 0)
11        ^LinkedBlockingQueue queue (LinkedBlockingQueue.)
12        running (atom true)
13        runner (Thread.
14          (fn []
15              (try-cause
16                  (while @running
17                     (let [r (.take queue)]
18                        (r)
19                        (swap! processed inc)))
20              (catch InterruptedException t
21                  (log-message "Event manager interrupted"))
22              (catch Throwable t
23                  (log-error t "Error when processing event")
24                  (halt-process! 20 "Error when processing an event"))
25                  )))]
26        (.setDaemon runner daemon?)
27        (.start runner)
28        (reify
29              EventManager
30              (add [this event-fn]
31                    ;; should keep track of total added and processed to know if this is finished yet
32                    (when-not @running
33                        (throw (RuntimeException. "Cannot add events to a shutdown event manager")))
34                    (swap! added inc)
35                    (.put queue event-fn)
36                    )
37        (waiting? [this]
38               (or (Time/isThreadWaiting runner)
39                      (= @processed @added)
40                ))
41          (shutdown [this]
42              (reset! running false)
43             (.interrupt runner)
44             (.join runner)
45             )
46             )))
```

调用event-manager方法后，将返回一个实现了EventManager协议的对象。通过该对象，用户可以向新创建的处理线程的LinkedBlockingQueue中添加事件，也可以关闭该处理线程。

4.4 even-sampler

在Storm中，我们采用采样的方式更新运行统计，Storm UI上显示的数据都是根据采样率进行估算的值。

even-sampler定义了一个均匀采样器，输入参数freq代表了采样频率。r为Java Random类型的

变量,代表一个随机数生成器。它每次随机生成一个介于[0, freq)之间的值,并将该值保存在target变量中。每次调用该函数时，都会将当前值与target进行比较，若相等则返回true。但要注意的是,只有当i大于等于freq时才重新置为0,这保证了采样的均匀性。even-sampler函数的定义如下：

```
1 (defn even-sampler [freq]
2     (let [freq (int freq)
3          start (int 0)
4          r (java.util.Random.)
5          curr (MutableInt. -1)
6          target (MutableInt. (.nextInt r freq))]
7          (with-meta
8              (fn []
9                  (let [i (.increment curr)]
10                     (when (>= i freq)
11                         (.set curr start)
12                         (.set target (.nextInt r freq))))
13                 (= (.get curr) (.get target)))
14             {:rate freq})))
```

函数mk-stats-sampler用于生成一个even-sampler：

```
(defn mk-stats-sampler [conf]
    (even-sampler (sampling-rate conf)))
```

函数sampleing-rate的值由TOPOLOGY-STATS-SAMPLE-RATE配置指定：

```
(defn sampling-rate [conf]
    (->> (conf TOPOLOGY-STATS-SAMPLE-RATE)
       (/ 1)
       int))
```

参数TOPOLOGY-STATS-SAMPLE-RATE在Storm中的默认值为0.05，此时sampling-rate函数的返回值应为20，即每隔20条消息采样一条消息。

4.5　ZooKeeper 工具类

Storm对ZooKeeper的访问是基于CuratorFramework的，本节将介绍如何访问ZooKeeper。

4.5.1　mk-client

mk-client函数用于创建一个ZooKeeper连接，并注册CuratorListener方法来监听ZooKeeper的事件（例如节点的创建、删除等），一旦有这样的事件发生，该监听器就会被调用。默认的监听器default-watcher只负责打印相关消息。该函数的定义如下：

```
1 (defnk mk-client [conf servers port :root "" :watcher default-watcher :auth-conf nil]
2     (let [fk (Utils/newCurator conf servers port root (when auth-conf (ZookeeperAuthInfo.
         auth-conf)))]
```

```
3         (.. fk
4             (getCuratorListenable)
5             (addListener
6                 (reify CuratorListener
7                     (^void eventReceived [this ^CuratorFramework _fk ^CuratorEvent e]
8                         (when (= (.getType e) CuratorEventType/WATCHED)
9                             (let [^WatchedEvent event (.getWatchedEvent e)]
10                                (watcher (zk-keeper-states (.getState event))
11                                    (zk-event-types (.getType event))
12                                    (.getPath event))))))))
```

- 第2行通过Utils的newCurator方法来创建一个Curator对象，该对象可用于对ZooKeeper进行数据操作。
- 第5行为该Curator添加了一个事件监听器，当ZooKeeper的状态发生变化时，watcher函数将会被调用。

4.5.2 create-node

create-node函数会利用CuratorFramework对象zk来创建节点并设置数据。在ZooKeeper中，存在三种类型的节点，具体如下所示。

- EPHEMERAL：临时节点，连接断开后节点即被删除。
- PERSISTENT：永久节点，连接断开后节点会继续保持。
- PERSISTENT_SEQUENTIAL：顺序永久节点，节点名字上带有序号，且ZooKeeper会保证该类节点是按照序号递增的方式被创建的。

相关代码如下：

```
1 (def zk-create-modes
2     {:ephemeral CreateMode/EPHEMERAL
3     :persistent CreateMode/PERSISTENT
4     :sequential CreateMode/PERSISTENT_SEQUENTIAL})
5
6 (defn create-node
7      ([^CuratorFramework zk ^String path ^bytes data mode]
8           (.. zk (create) (withMode (zk-create-modes mode)) (withACL ZooDefs$Ids/OPEN_ACL
               _UNSAFE) (forPath (normalize-path path) data)))
9 ([^CuratorFramework zk ^String path ^bytes data]
10    (create-node zk path data :persistent)))
```

4.5.3 get-data

该函数利用CuratorFramework的getData方法获取数据。值得注意的是，该函数可以传入watch?参数，以指明是否对路径path进行监听。当数据更新时，注册的watcher回调方法会被调用。回调方法被调用后就失效，如果想继续监听，需要在下次调用get-data函数时进行重新设置。该函数的代码如下：

```
(defn get-data [^CuratorFramework zk ^String path watch?]
    (let [path (normalize-path path)]
      (try-cause
        (if (exists-node? zk path watch?)
            (if watch?
                (.. zk (getData) (watched) (forPath path))
                (.. zk (getData) (forPath path))))
    (catch KeeperException$NoNodeException e
        ;; this is fine b/c we still have a watch from the successful exists call
        nil ))))
```

通过watcher的回调，系统可以尽快地得到数据的变化通知。例如，当Topology的状态变动非活跃时，Worker可以根据该变化来调整自身执行的Executor。但是，watcher的回调方法不能保证一定会被调用到，通常需要与定时器结合使用。

4.5.4　进程内启动ZooKeeper

mk-inprocess-zookeeper函数用于在一个进程内打开ZooKeeper服务，主要用于LocalCluster模式。其基本思路为利用NIOServerCnxnFactory来管理ZooKeeper连接。该工厂方法需要绑定到一个端口号，Storm从2000开始探测可以使用的端口号，并最终由mk-inprocess-zookeeper函数返回ZooKeeper启动所使用的端口号。该函数的代码如下：

```
1 (defnk mk-inprocess-zookeeper [localdir :port nil]
2     (let [localfile (File. localdir)
3         zk (ZooKeeperServer. localfile localfile 2000)
4         [retport factory] (loop [retport (if port port 2000)]
5                             (if-let [factory-tmp (try-cause (NIOServerCnxnFactory/create
                                Factory retport 60) ;; 60 is the default maxclientcnxns
6                             (catch BindException e
7                             (when (> (inc retport) (if port port 65535))
8                             (throw (RuntimeException. "No port is available to launch an
                                inprocess zookeeper."))))))]
9         [retport factory-tmp]
10        (recur (inc retport))))]
11    (log-message "Starting inprocess zookeeper at port " retport " and dir " localdir)
12    (.startup factory zk)
13    [retport factory]
14    ))
```

4.6　LocalState

LocalState定义了一个轻量级的、可持久化的K/V数据库。它的效率不高，每次读写都要进行磁盘操作，所以一般只能用于读写次数不多的场景。在Storm中，LocalState主要在Supervisor和Worker中使用。它的实现如下：

```
1 public class LocalState {
2   private VersionedStore _vs;
3
```

```
4     public LocalState(String backingDir) throws IOException {
5         _vs = new VersionedStore(backingDir);
6     }
7
8     public synchronized Map<Object, Object> snapshot() throws IOException {
9         String latestPath = _vs.mostRecentVersionPath();
10        if(latestPath==null) return new HashMap<Object, Object>();
11        return (Map<Object, Object>) Utils.deserialize(FileUtils.readFileToByteArray(new File
              (latestPath)));
12    }
13
14    public Object get(Object key) throws IOException {
15        return snapshot().get(key);
16    }
17
18    public synchronized void put(Object key, Object val) throws IOException {
19        Map<Object, Object> curr = snapshot();
20        curr.put(key, val);
21        persist(curr);
22    }
23
24    public synchronized void remove(Object key) throws IOException {
25        Map<Object, Object> curr = snapshot();
26        curr.remove(key);
27        persist(curr);
28    }
29
30    private void persist(Map<Object, Object> val) throws IOException {
31        byte[] toWrite = Utils.serialize(val);
32        String newPath = _vs.createVersion();
33        FileUtils.writeByteArrayToFile(new File(newPath), toWrite);
34        _vs.succeedVersion(newPath);
35        _vs.cleanup(4);
36    }
37}
```

这个类使用了VersionedStore对象，该对象主要实现一个简单的带版本的文件管理器。它能够生成新的版本号（基于系统当前时间），同时给每个成功写入的本地文件添加一个对应的以.version为后缀的文件作为其版本信息，还能够管理版本的保存和删除。下面简要介绍该类中涉及的方法。

- LocalState构造函数需要指定一个root路径，同时会将这个路径作为VersionedStore的root路径。
- snapshot方法获取当前root路径下最新版本文件中记录的信息的快照，它不会修改文件内容。
- get方法获取最新版本文件中所包含的由键指定的内容。
- put方法将新的键值对放入当前最新版本的内容中，同时创建一个最新版本的文件保存更改过后的内容。
- remove方法将当前最新版本的内容中由键指定的值删除，同时创建一个最新版本的文件保存更改过后的内容。

- persist方法是真正实现保存更改后内容的方法。它将内容序列化，通过VersionedStore创建一个新版本的文件，将序列化后的内容保存到该文件中，然后调用succeedVersion方法为该文件创建版本信息，并调用cleanup方法清除老版本的文件，保证只留下最新的4个版本的文件。

4.7 ClusterState

ClusterState协议定义了一系列用于对ZooKeeper进行操作的方法。mk-distributed-cluster-state函数则返回一个实现了该协议的对象，该对象的基本方法都是利用Zookeeper.clj中定义的方法实现的，这里主要讨论回调函数的注册和调用，相关代码如下：

```
1 (defn mk-distributed-cluster-state [conf]
2     (let [zk (zk/mk-client conf (conf STORM-ZOOKEEPER-SERVERS) (conf STORM-ZOOKEEPER-PORT) :
          auth-conf conf)]
3         (zk/mkdirs zk (conf STORM-ZOOKEEPER-ROOT))
4         (.close zk))
5     (let [callbacks (atom {})
6          active (atom true)
7          zk (zk/mk-client conf
8              (conf STORM-ZOOKEEPER-SERVERS)
9              (conf STORM-ZOOKEEPER-PORT)
10             :auth-conf conf
11             :root (conf STORM-ZOOKEEPER-ROOT)
12             :watcher (fn [state type path]
13                (when @active
14                    (when-not (= :connected state)
15                          (log-warn "Received event " state ":" type ":" path " with
                                disconnectedZookeeper."))
16                    (when-not (= :none type)
17                          (doseq [callback (vals @callbacks)]
18                              (callback type path))))
19                    ))]
20        (reify
21            ClusterState
22              (register [this callback]
23                  (let [id (uuid)]
24                       (swap! callbacks assoc id callback)
25                       id
26                       ))
27              (unregister [this id]
28                  (swap! callbacks dissoc id))
29              )))
```

- 第2~4行根据配置项中定义的ZooKeeper服务器、端口号和认证方式，调用mk-client方法创建一个新的ZooKeeper客户端，并在第3行创建根目录。
- 第5行定义了用来存储所有回调函数的callbacks变量。
- 第7~19行重新创建ZooKeeper的客户端，不过这次是将第11行传入的:root作为该客户端的

根目录，该客户端的其他操作都是相对于该根目录进行的。预先创建根目录有助于提高ZooKeeper的访问性能。

- 第12~19行定义调用mk-client时传入的回调函数，其中第17~18行将依次调用callbacks中存储的函数。
- 第22~26行调用register方法来更新callbacks变量。于是，新注册的方法便可以被ZooKeeper的watcher方法回调。
- 第27~28行取消了一个回调的注册。

通过对callbacks变量进行操作，就可实现动态地增减回调函数这一目标。

4.8 StormClusterState

StormClusterState协议定义了与Storm相关的ZooKeeper操作，例如读取Topology的任务分配、向ZooKeeper中发送心跳信息等。

mk-storm-cluster-state函数用于创建一个实现了StormClusterState协议的对象。本节主要讨论其中有关ZooKeeper使用的内容。该函数的代码如下：

```
1 (defn mk-storm-cluster-state [cluster-state-spec]
2   (let [[solo? cluster-state] (if (satisfies? ClusterState cluster-state-spec)
3       [false cluster-state-spec]
4       [true (mk-distributed-cluster-state cluster-state-spec)])
5       assignment-info-callback (atom {})
6       supervisors-callback (atom nil)
7       assignments-callback (atom nil)
8       storm-base-callback (atom {})
9       state-id (register
10          cluster-state
11          (fn [type path]
12              (let [[subtree & args] (tokenize-path path)]
13                   (condp = subtree
14                       ASSIGNMENTS-ROOT (if (empty? args)
15                           (issue-callback! assignments-callback)
16                           (issue-map-callback! assignment-info-callback (first args)))
17                       SUPERVISORS-ROOT (issue-callback! supervisors-callback)
18                       STORMS-ROOT (issue-map-callback! storm-base-callback (first args))
19                       ;; this should never happen
20                       (halt-process! 30 "Unknown callback for subtree " subtree args)
21                       )
22                   )))]
23      (doseq [p [ASSIGNMENTS-SUBTREE STORMS-SUBTREE SUPERVISORS-SUBTREE WORKERBEATS-SUBTREE
          ERRORS-SUBTREE]]
24      (mkdirs cluster-state p))
```

- 第2行根据传入参数是否实现了ClusterState协议，来判断是否应该调用mk-distributed-cluster-state来创建ClusterState对象以操作ZooKeeper。若传入的cluster-state-spec未实现ClusterState协议，则说明该ClusterState对象仅供其自己使用，退出时需要断开与ZooKeeper的连接。

- 第5~8行分别定义了一些回调函数的集合，用于监听不同的目录。
- 第9~21行调用ClusterState的reigster方法将一个回调函数注册到ClusterState的callbacks变量中。第12~21行定义了该回调函数，它是根据watcher回调函数发回的路径来调用相应的函数。例如，若发回的路径为SUPERVISORS-ROOT，则调用supervisors-callback所对应的回调方法。issue-callback!函数在执行完supervisors-callback方法后会将回调函数重置为null。

例如，对于assignments方法的实现：

```
(assignments [this callback]
    (when callback
        (reset! assignments-callback callback))
    (get-children cluster-state ASSIGNMENTS-SUBTREE (not-nil? callback)))
```

若callback函数不为null，首先将assignments-callback重置为传入的callback函数，然后在调用get-children函数时将最后一个参数设为true，表示客户端需要监测该路径下数据的变化，一旦路径ASSIGNMENTS-SUBTREE下的数据发生变化，ZooKeeper将通知客户端来执行回调函数。

这样做是很有意义的，当Nimbus重新分配任务时，各个Supervisor需要在第一时间获得通知，并根据重新分配的任务来调整自身运行。

由于ZooKeeper中回调方法的调用是不被保证的，通常还需要额外的线程来定期获取数据，以实现数据同步。

第 5 章

通信机制

在进程之间，Storm采用ZMQ进行通信。ZMQ是一个开源的、跨语言的网络通信库，它非常简洁，运行性能高，使用起来也十分灵活。读者可以参考以下链接来学习ZMQ：

- http://zguide.zeromq.org/
- http://zguide.zeromq.org/page:all#Core-Messaging-Patterns

对于同一进程内部的诸多线程，Storm则采用Disruptor Queue来完成它们之间的通信。通过阅读下面的链接，你将对LAMX Disruptor的特点和原理产生更加深刻的认识。

- http://lmax-exchange.github.io/disruptor/
- http://ifeve.com/disruptor/
- http://www.cnblogs.com/fxjwind/p/3180073.html
- http://blog.sina.com.cn/s/blog_68ffc7a4010150yl.html

在本章中，我们主要介绍Storm中进程间以及进程内线程之间的通信方法。

5.1 进程间通信

对于不同的系统模式，进程之间通信的具体实现方式也是不同的。

在分布式模式下，如前所述，Storm将采用ZMQ进行通信。

而在LocalCluster模式下，如果参数storm.local.mode.zmq的值为false（默认为false），Storm将会采用线程模拟进程的方式启动服务，即通过一个LinkedBlockingQueue来模拟进程间通信。如果参数storm.local.mode.zmq的值为true，则如同分布式模式一样，使用ZMQ来进行通信。

Storm首先定义了一些必要的通信协议，以便更加灵活地支持以上这两种模式。而具体的通信过程，则是对协议的进一步实现和使用。

5.1.1 进程间通信协议

用于进程间通信的协议主要包括Connection和Context。

Connection协议定义了3个方法，用于接收和发送消息：

```
(defprotocol Connection
    (recv-with-flags [conn flags])
```

```
    (send [conn task message])
    (close [conn])
    )
```

recv-with-flags方法用来接收消息，参数flags表示是否为阻塞方式。send方法send用来向某一个Task发送消息，close方法用于关闭一个连接。

Context协议定义了创建和关闭Socket连接的方法：

```
(defprotocol Context
    (bind [context storm-id port])
    (connect [context storm-id host port])
    (term [context])
    )
```

在上述代码中，bind方法用于绑定到一个端口号并返回一个Socket对象，该Socket对象主要负责接收消息，而connect方法则用于连接到一个端口号并返回该端口号，主要负责发送消息。term方法用来关闭连接。

5.1.2 LocalCluster模式实现

在LocalCluster模式下，系统默认（storm.local.mode.zmq为false）采用一个进程来模拟Storm集群环境，Nimbus、Supervisor、Worker等都是该进程中的一个线程。所以，这种情况下的通信实际上是进程内通信。Storm使用LinkedBlockingQueue来模拟消息的发送和接收。接下来，我们介绍LocalCluster模式下Storm是如何实现Connection及Context协议的。

1. LocalConnection

LocalConnection类型实现了协议Connection，它的成员变量包括storm-id、port、queues-map、lock以及queue，其中queues-map为一个哈希表，键由storm-id以及端口号port连接而成，值为一个LinkedBlockingQueue队列。成员变量queue为接收队列。LocalConnection的代码如下：

```
1  (deftype LocalConnection [storm-id port queues-map lock queue]
2      Connection
3      (recv-with-flags [this flags]
4          (when-not queue
5              (throw (IllegalArgumentException. "Cannot receive on this socket")))
6          (if (= flags 1)
7              (.poll queue)
8              (.take queue)))
9      (send [this task message]
10         (let [send-queue (add-queue! queues-map lock storm-id port)]
11             (.put send-queue [task message])
12             ))
13     (close [this]
14        ))
```

- 第3~8行实现recv-with-flags函数，该函数的第一个参数为this指针，即协议中的conn参数为LocalConnection类型。flags为1表示以非阻塞方式获取消息，否则会以阻塞方式获取消息。

- 第9~12行实现send函数。第10行通过add-queue!函数获得一个发送消息的队列。第11行向队列里发送一条消息，该消息由两部分构成，即TaskId以及消息的实际内容。

add-queue!的实现如下：

```
(defn add-queue! [queues-map lock storm-id port]
    (let [id (str storm-id "-" port)]
      (locking lock
         (when-not (contains? @queues-map id)
             (swap! queues-map assoc id (LinkedBlockingQueue.))))
    (@queues-map id)))
```

queues-map的键由storm-id与port连接而成，该函数首先根据传入的storm-id以及port计算得到queues-map的键并将其存储到变量id中。若queues-map里没有id，函数就创建一个新的LinkedBlockingQueue对象将其与id关联。函数的末尾返回id所对应的队列，该队列或者是新创建的或者是已经存在的。

5

2. LocalContext

LocalContext类型中含有域queues-map以及用于操作queues-map的锁对象lock。该类型实现了Context协议，其代码如下：

```
(deftype LocalContext [queues-map lock]
    Context
    (bind [this storm-id port]
        (LocalConnection. storm-id port queues-map lock (add-queue! queues-map lock storm-id port)))
    (connect [this storm-id host port]
        (LocalConnection. storm-id port queues-map lock nil)
        )
    (term [this]
    ))
```

在上述代码中，bind函数主要用于接收消息，它会根据输入的storm-id和port创建一个LocalConnection对象，并传入一个LinkedBlockingQueue队列作为接收队列。connect方法则主要用于发送消息，它同样会创建LocalConnection对象，只不过其接收队列为nil。

要创建一个LocalContext对象，需使用mk-local-context函数。这里，queues-map被初始化为atom哈希类型，而lock锁对象则是一个新创建的Object对象：

```
(defn mk-local-context []
    (LocalContext. (atom {}) (Object.)))
```

5.1.3 分布式模式实现

在分布式模式下，Storm采用ZMQ来进行进程间通信。接下来，我们介绍一下分布式模式下Storm是如何实现通信协议的。

1. ZMQConnection

ZMQConnection类实现了Connection协议，其代码如下：

```
(deftype ZMQConnection [socket ^ByteBuffer bb]
    Connection
    (recv-with-flags [this flags]
      (let [part1 (mq/recv socket flags)]
          (when part1
          (when-not (mq/recv-more? socket)
              (throw (RuntimeException. "Should always receive two-part ZMQ messages")))
          (parse-packet part1 (mq/recv socket)))))
    (send [this task message]
        (.clear bb)
        (.putShort bb (short task))
        (mq/send socket (.array bb) NOBLOCK-SNDMORE)
        (mq/send socket message ZMQ/NOBLOCK)) ;; TODO: how to do backpressure if doing noblock?... need
          to only unblock if the target disappears
    (close [this]
        (.close socket)
        ))
(defn mk-connection [socket]
    (ZMQConnection. socket (ByteBuffer/allocate 2)))
```

成员变量socket表示ZMQ的连接字符串。bb为预先分配的ByteBuffer空间，用来存储已被序列化的端口号。

ZMQ会分两次发送消息：第一次发送TaskId，第二次则为具体的消息内容。接收时同样先接收TaskId，然后接收具体内容。

mk-connection函数用来实例化一个ZMQConnection对象。

2. ZMQContext

ZQMContext类实现了Context协议，其代码如下：

```
(deftype ZMQContext [context linger-ms hwm local?]
    Context
    (bind [this storm-id port]
        (-> context
            (mq/socket mq/pull)
            (mq/set-hwm hwm)
            (mq/bind (get-bind-zmq-url local? port))
            mk-connection
           ))
    (connect [this storm-id host port]
         (-> context
             (mq/socket mq/push)
             (mq/set-hwm hwm)
             (mq/set-linger linger-ms)
             (mq/connect (get-connect-zmq-url local? host port))
            mk-connection))
    (term [this]
        (.term context))
    ZMQContextQuery
    (zmq-context [this]
      context))
```

- context为ZMQContext对象。
- linger-ms表示调用ZMQ Socket的关闭方法term后未发送消息的等待时间。若超出该时间，未发送的消息将被丢弃。该变量的默认值为5秒，由zmq.linger.millis配置参数设置。
- hwm表示ZMQ发送队列的高水平线。若发送队列里面的消息个数超过hwm，新来的消息可能会被丢弃。该变量的默认值为0，即没有限制。
- local?表示系统是否运行在单机环境下，它被用于LocalCluster模式的ZMQ中，即将storm.local.mode.zmq设置为true。
- bind方法设置了ZMQ的Socket参数。注意这里Socket模式设置为pull类型，表示返回的Socket主要用于接收消息。Bind方法会调用get-bind-zmq-url函数来获得Socket连接字符串，如下面的代码所示，协议的主机名部分为*，表示从当前主机对应的端口接收消息：

```
(defn get-bind-zmq-url [local? port]
    (if local?
        (str "ipc://" port ".ipc")
        (str "tcp://*:" port)))
```

5

类似地，connect方法也会设置ZMQ的Socket参数，只不过它将Socket模式设置为push。该函数返回的Socket主要用于发送消息。

connect方法会调用get-connect-zmq-url函数以获得Socket连接字符串。从下面的代码可以看出，get-connect-zmq-url需要给定主机和端口号，表示将消息发送到目标主机host的端口号port上：

```
(defn get-connect-zmq-url [local? host port]
    (if local?
        (str "ipc://" port ".ipc")
        (str "tcp://" host ":" port)))
```

5.1.4 协议使用

Storm会调用launch-receive-thread!函数来启动Worker所对应的消息接收线程。该函数在后台启动接收线程，同时返回一个用于终止线程接收的函数。其代码如下：

```
1 (defnk launch-receive-thread!
2     [context storm-id port transfer-local-fn max-buffer-size
3         :daemon true
4         :kill-fn (fn [t] (System/exit 1))
5         :priority Thread/NORM_PRIORITY]
6     (let [max-buffer-size (int max-buffer-size)
7          vthread (async-loop
8            (fn []
9                (let [socket (msg/bind context storm-id port)]
10                    (fn []
11                       (let [batched (ArrayList.)
12                             init (msg/recv socket)]
13                       (loop [[task msg :as packet] init]
14                           (if (= task -1)
```

```
15                              (do (log-message "Receiving-thread:[" storm-id ", " port "]
                                    received shutdown notice")
16                                  (.close socket)
17                                  nil )
18                              (do
19                                  (when packet (.add batched packet))
20                                  (if (and packet (< (.size batched) max-buffer-size))
21                                  (recur (msg/recv-with-flags socket 1))
22                                    (do (transfer-local-fn batched)
23                                        0 ))))))))
24                :factory? true
25                :daemon daemon
26                :kill-fn kill-fn
27                :priority priority)]
28    (fn []
29        (let [kill-socket (msg/connect context storm-id "localhost" port)]
30            (log-message "Shutting down receiving-thread: [" storm-id ", " port "]")
31            (msg/send kill-socket
32                -1
33                (byte-array []))
34            (log-message "Waiting for receiving-thread:[" storm-id ", " port "] to die")
35            (.join vthread)
36            (.close kill-socket)
37            (log-message "Shutdown receiving-thread: [" storm-id ", " port "]")
38            ))))
```

- 在第2行的接收参数中，context为上下文对象，它可以是LocalContext对象或者ZMQContext对象。transfer-local-fn对应Worker的一个接收函数，用来处理从队列中接收到的消息，即将消息分发到Worker的各个Disruptor Queue中。max-buffer-size表示最少收到多少消息之后调用transfer-local-fn函数，它由topology.receiver.buffer.size配置项确定，默认值为8。
- 第7～38行通过调用async-loop方法启动了一个接收线程。第9行调用协议的bind方法返回一个Socket对象。对于不同的Context对象，返回的Socket是不同的。目前，Storm发送与接收消息的协议如下。
- 普通消息由两部分构成，第一个部分表示消息的来源TaskId，第二个部分为消息的具体内容。
- 接收端Task以阻塞方式接收第一条消息，然后换由非阻塞方式接收其他消息，直到缓存的消息数目超过max-buffer-size的值；
- 若收到消息的TaskId为−1，表示收到关闭接收线程的信号。第16行关闭Socket，第17行返回nil，表示退出线程的循环体。
- 第11行初始化batched变量，用于存储接收到的消息。
- 第12行以阻塞模式调用msg/recv函数。
- 第19行中，若消息不为null，则将消息存储到batched缓存中。
- 第20~23行，若收到的消息不为空，同时缓存的消息数目少于max-buffer-size，则继续以非阻塞模式接收消息。否则，调用transfer-local-fn函数处理目前缓存的消息，同时返

回0表示终止第13行定义的循环。接下来，async-loop中的循环会重新调用第10行定义的函数，开始新一轮的消息接收。

- 第28~38行定义了一个用来杀掉接收线程的函数。它首先向线程发送一条消息，消息对应的TaskId为-1，然后调用线程的join方法以及Socket的close方法，最终杀掉线程。

Worker中的launch-receive-thread方法用来调用上面的函数，其代码如下：

```
(defn launch-receive-thread [worker]
    (log-message "Launching receive-thread for " (:assignment-id worker) ":" (:port worker))
    (msg-loader/launch-receive-thread!
       (:mq-context worker)
       (:storm-id worker)
       (:port worker)
       (:transfer-local-fn worker)
       (-> worker :storm-conf (get TOPOLOGY-RECEIVER-BUFFER-SIZE))
       :kill-fn (fn [t] (halt-process! 11))))
```

5.2 进程内通信

Disruptor Queue是用于线程之间通信的高效通信队列。为了提高性能，Storm中所有的进程内通信都会基于Disruptor Queue来实现。下面我们就来介绍Storm使用Disruptor Queue的细节。

5.2.1 Disruptor Queue的使用

在Storm中，我们使用Disruptor Queue类来对Disruptor Queue的使用进行封装，本节简要介绍一下该类。

1. 成员变量分析

DisruptorQueue类对LAMX Disruptor框架进行了封装，Storm使用这个类来发送和接收消息。该类的定义如下：

```
public class DisruptorQueue {
   static final Object FLUSH_CACHE = new Object();
   static final Object INTERRUPT = new Object();

   RingBuffer<MutableObject> _buffer;
   Sequence _consumer;
   SequenceBarrier _barrier;

   // TODO: consider having a threadlocal cache of this variable to speed up reads?
   volatile boolean consumerStartedFlag = false;
   ConcurrentLinkedQueue<Object> _cache = new ConcurrentLinkedQueue();
}
```

- _cache为一个ConcurrentLinkedQueue对象。若队列还未启动用户就已经发送来了消息，那么这些消息就会被临时存放在该变量中。

- FLUSH_CATCH为特殊类型的对象。系统会在队列启动时向队列中插入该对象，而收到该对象后，对_cache中的缓存数据进行处理。
- _INTERRUPT为另外一个信号对象，收到该对象时消息的接收循环将被结束。
- _buffer为RingBuffer对象，对应于Disruptor队列的存储。
- _consumer为Sequence对象，对应于Disruptor队列的接收端。
- _barrier为SequenceBarrier对象，用于等待并获取可读数据。
- consumerStartedFlag用来表示接收端是否开始，当接收端尚未开始时，会将数据放于成员变量_cache中。

2. publish函数

publish函数用于向RingBuffer对象_buffer中存储数据，其代码如下所示：

```
1  public void publish(Object obj, boolean block) throws InsufficientCapacityException {
2
3     if(consumerStartedFlag) {
4         final long id;
5         if(block) {
6                 id = _buffer.next();
7         } else {
8                 id = _buffer.tryNext(1);
9         }
10        final MutableObject m = _buffer.get(id);
11        m.setObject(obj);
12        _buffer.publish(id);
13    } else {
14        _cache.add(obj);
15        if(consumerStartedFlag) flushCache();
16    }
17 }
18 public void publish(Object obj) {
19     try {
20          publish(obj, true);
21     } catch (InsufficientCapacityException ex) {
22          throw new RuntimeException("This code should be unreachable!");
23     }
24 }
```

若接收端已经启动，则调用RingBuffer对象的next或者tryNext方法来获取下一个存储位置的id。然后，代码的第10~12行会设置并调用RingBuffer对象的publish函数，将要发送的消息对象存储到RingBuffer中。目前，Storm都采用阻塞的方法发布消息，即block参数为true。否则block参数为false，消息将被存入_cache对象中，然后判断是否应调用flushCache操作，该操作会向队列中发布一个FLUSH_CACHE对象。

3. consumeBatchToCursor函数

consumeBatchToCursor函数会访问RingBuffer对象并获取一组可读的数据，它通常被放在一个循环中反复执行以获取数据，用来处理RingBuffer中存储的数据，其参数cursor为目前RingBuffer的最大可读游标，handler为EventHandler对象，是对收到消息的回调函数。例如，由Worker的函

数mk-transfer-tuples-handler创建的一个新函数通过不断调用consumeBatchToCursor方法来获得发送至该Worker的数据。consumeBatchToCursor的代码如下：

```
1 private void consumeBatchToCursor(long cursor, EventHandler<Object> handler) {
2     for(long curr = _consumer.get() + 1; curr <= cursor; curr++) {
3         try {
4             MutableObject mo = _buffer.get(curr);
5             Object o = mo.o;
6             mo.setObject(null);
7             if(o==FLUSH_CACHE) {
8                 Object c = null;
9                 while(true) {
10                    c = _cache.poll();
11                    if(c==null) break;
12                    else handler.onEvent(c, curr, true);
13                }
14            } else if(o==INTERRUPT) {
15                throw new InterruptedException("Disruptor processing interrupted");
16            } else {
17                handler.onEvent(o, curr, curr == cursor);
18            }
19        } catch (Exception e) {
20            throw new RuntimeException(e);
21        }
22     }
23    //TODO: only set this if the consumer cursor has changed?
24    _consumer.set(cursor);
25 }
```

- 在第2行中，_consumer.get()方法可获得上次读到的位置。consumeBatchToCursor函数返回一组数据，这组数据的区间为[_consumer.get()+1, cursor]。
- 第4~6行获取相应的消息对象。
- 第7~13行处理接收端尚未启动时发送的消息，这些消息被放入_cache变量中。当接收端启动后，将优先发送缓存_cache中的消息。
- 第17行调用handler的onEvent函数。若curr与cursor相同，则表示该Batch结束，Worker会根据Batch是否结束来决定是否将缓存的消息发送出去。
- 第24行重新设置_consumer所对应的游标。

consumeBatchWhenAvailable函数会等待_barrier的函数返回，并调用consumeBatchToCursor函数来处理消息，相关代码如下：

```
public void consumeBatchWhenAvailable(EventHandler<Object> handler) {
    try {
        final long nextSequence = _consumer.get() + 1;
        final long availableSequence = _barrier.waitFor(nextSequence, 10, TimeUnit.MILLISECONDS);
        if(availableSequence >= nextSequence) {
            consumeBatchToCursor(availableSequence, handler);
        }
    } catch (AlertException e) {
        throw new RuntimeException(e);
```

```
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
```

这里要注意，目前waitFor函数最多等待10毫秒。

consumeBatch函数对应于非阻塞情况，它通过getCursor的返回值直接调用comsumeBatchToCursor函数。consumeBatch主要用于Spout的消息处理线程，由于该线程中还要执行一些其他操作，因此必须进行非阻塞的调用。其代码如下：

```
public void consumeBatch(EventHandler<Object> handler) {
    consumeBatchToCursor(_barrier.getCursor(), handler);
}
```

5.2.2 DisruptorQueue的Clojure处理器

disruptor.clj文件中定义了一些封装了DisruptorQueue方法的方法，这里主要介绍consumer-loop*函数和coljure-handler函数。

consume-loop*函数用于启动一个消费者线程，该线程会不断调用DisruptorQueue的consume-batch-when-available方法。而且线程在启动后，还会执行consume- start!操作，设置DisruptorQueue中的consumerStartedFlag标志，以表明该队列可以接收数据了：

```
(defnk consume-loop* [^DisruptorQueue queue handler :kill-fn (fn [error] (halt-process! 1 "Async loop
    died!"))]
    (let [ret (async-loop
         (fn []
            (consume-batch-when-available queue handler)
            0 )
            :kill-fn kill-fn
            )]
    (consumer-started! queue)
    ret
    ))
(defn consume-batch-when-available [^DisruptorQueue queue handler]
    (.consumeBatchWhenAvailable queue handler))
```

clojure-handler函数用来实例化一个EventHandler对象，其输入参数为一个函数（afn），且在实现onEvent方法时将调用这个传入的函数。clojure-handler的代码如下：

```
(defn clojure-handler [afn]
    (reify com.lmax.disruptor.EventHandler
        (onEvent [this o seq-id batchEnd?]
           (afn o seq-id batchEnd?)
           )))
```

Worker中的mk-transfer-tuples-handler函数将使用clojure-handler来定义一个EventHandler，这将在第9章中介绍。

第 6 章

Nimbus

6

Nimbus可以说是Storm中最核心的部分，它的主要功能有两个，具体如下所示。

- 对Topology的任务进行分配调度。
- 接收用户的命令并做相应的处理，例如Topology的提交（submit）、杀死（kill）、激活（activate）、暂停（deactivate）以及重新调度（rebalance）。

Nimbus本身里基于Thrift框架（http://thrift.apache.org/）实现的，它使用了Thrift的THsHaServer服务。THsHaServer即半同步半异步服务模式，它使用一个单独的线程来处理网络I/O，使用一个独立的工作者线程池来处理消息，大大提高了消息的并发处理能力。

6.1 Nimbus 服务接口定义

Nimbus服务接口是用Thrift的语法编写的，它定义了Nimbus提供的所有服务，其代码如下：

```
1 service Nimbus {
2     void submitTopology(1: string name, 2: string uploadedJarLocation, 3: string jsonConf, 4:
          StormTopology topology) throws (1: AlreadyAliveException e, 2: InvalidTopologyException
          ite);
3     void submitTopologyWithOpts(1: string name, 2: string uploadedJarLocation, 3: string jsonConf,
          4: StormTopology topology, 5: SubmitOptions options) throws (1: AlreadyAliveException e, 2:
          InvalidTopologyException ite);
4     void killTopology(1: string name) throws (1: NotAliveException e);
5     void killTopologyWithOpts(1: string name, 2: KillOptions options) throws (1: NotAliveException e);
6     void activate(1: string name) throws (1: NotAliveException e);
7     void deactivate(1: string name) throws (1: NotAliveException e);
8     void rebalance(1: string name, 2: RebalanceOptions options) throws (1: NotAliveException e, 2:
          InvalidTopologyException ite);
9
10  // need to add functions for asking about status of storms, what nodes they're running on, looking
      at task logs
11
12  string beginFileUpload();
13  void uploadChunk(1: string location, 2: binary chunk);
14  void finishFileUpload(1: string location);
15
16  string beginFileDownload(1: string file);
17  //can stop downloading chunks when receive 0-length byte array back
18  binary downloadChunk(1: string id);
```

```
19
20  // returns json
21  string getNimbusConf();
22  // stats functions
23  ClusterSummary getClusterInfo();
24  TopologyInfo getTopologyInfo(1: string id) throws (1: NotAliveException e);
25  //returns json
26  string getTopologyConf(1: string id) throws (1: NotAliveException e);
27  StormTopology getTopology(1: string id) throws (1: NotAliveException e);
28  StormTopology getUserTopology(1: string id) throws (1: NotAliveException e);
```

- 第2~3行定义了提交Topology的方法。
- 第4~5行定义了杀死Topology的方法。
- 第6行定义了激活某个Topology的方法。
- 第7行定义了暂停某个Topology的方法。
- 第8行定义了重新调度Topology任务的方法。
- 第12~14行定义了上传文件的方法。
- 第16~18行定义了下载文件的方法。
- 第21行定义了获取Nimbus服务所使用的Storm配置项的方法，它返回的是一个经过JSON序列化处理的字符串。
- 第23行定义了获取当前集群总体统计信息的方法。
- 第24行定义了获取某个Topology的总体统计信息的方法。
- 第26行定义了获取某个Topology的Storm配置项的方法。
- 第27行定义了获取系统Topology的方法。系统Topology是指在用户提交的Topology基础上添加acker、metric等系统定义bolt后形成的Topology。
- 第28行定义了获取用户提交的Topology的方法。

由于Storm运行在JVM上，前面定义的结构需使用Thrift转换成对应的Java代码。在得到Nimbus.java文件中，接口Iface的定义如下：

```
1 public interface Iface {
2     public void submitTopology(String name, String uploadedJarLocation, String jsonConf, Storm
          Topology topology) throws AlreadyAliveException, InvalidTopologyException, org.apache.
          thrift7.TException;
3
4     public void submitTopologyWithOpts(String name, String uploadedJarLocation, String jsonConf,
          StormTopology topology, SubmitOptions options) throws AlreadyAliveException, Invalid
          Topology Exception, org.apache.thrift7.TException;
5
6     public void killTopology(String name) throws NotAliveException, org.apache.thrift7.TException;
7
8     public void killTopologyWithOpts(String name, KillOptions options) throws NotAliveException,
         org.apache.thrift7.TException;
9
10    public void activate(String name) throws NotAliveException, org.apache.thrift7.TException;
11
```

```
12    public void deactivate(String name) throws NotAliveException, org.apache.thrift7.TException;
13
14    public void rebalance(String name, RebalanceOptions options) throws NotAliveException,
          InvalidTopologyException, org.apache.thrift7.TException;
15
16    public String beginFileUpload() throws org.apache.thrift7.TException;
17
18    public void uploadChunk(String location, ByteBuffer chunk) throws org.apache.thrift7.TException;
19
20    public void finishFileUpload(String location) throws org.apache.thrift7.TException;
21
22    public String beginFileDownload(String file) throws org.apache.thrift7.TException;
23
24    public ByteBuffer downloadChunk(String id) throws org.apache.thrift7.TException;
25
26    public String getNimbusConf() throws org.apache.thrift7.TException;
27
28    public ClusterSummary getClusterInfo() throws org.apache.thrift7.TException;
29
30    public TopologyInfo getTopologyInfo(String id) throws NotAliveException, org.apache.thrift7.
          TException;
31
32    public String getTopologyConf(String id) throws NotAliveException, org.apache.thrift7.TException;
33
34    public StormTopology getTopology(String id) throws NotAliveException, org.apache.thrift7.
          TException;
35
36    public StormTopology getUserTopology(String id) throws NotAliveException, org.apache.thrift7.
          TException;
37  }
```

Nimbus提供了该接口的具体实现，我们会在后面详细地介绍这些方法。

6.2 Nimbus 相关的数据结构

Nimbus中用到的数据结构主要有两大类：Java定义的数据结构以及Clojure定义的数据结构。Java定义的数据结构主要用于任务分配，而Clojure定义的数据结构则主要来充当一些Storm的元数据。

6.2.1 Java 数据结构

图6-1给出了Nimbus中用到的几个Java数据结构间的类关系图。

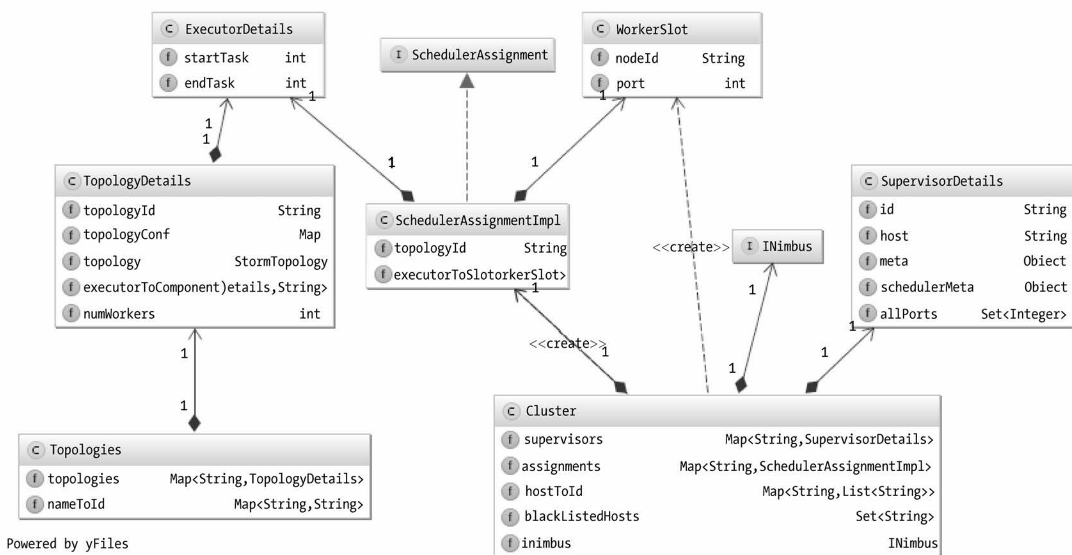


图6-1　Java数据结构

下面简要介绍图中各个对象的功能。

- ExecutorDetails对象记录了每个Executor所对应的startTask和endTask，这样定义是为了保证Storm在计算Executor时，每个都Executor都是一个连续的任务集合。
- TopologyDetails对象记录了每个Topology的信息，包括topologyId、Topology使用过的配置项、Topology对象本身、每个Executor到所属组件的以及numWorkers等信息。
- Topologies对象包含了一组Topology，它定义了一个<topology-id, TopologyDetails>集合以及一个<topology-name, topology-id>集合。
- SupervisorDetails对象记录了当前Supervisor的状态，包括id、host、meta和allports等信息。
- WorkerSlot对象定义了一个可用资源，它包含nodeId和port两个成员变量，其中nodeId就是前面提到的supervisor-id，它实际上是指某个Supervisor机器上的某个端口号。
- SchedulerAssignmentImpl对象定义了当前Topology的任务分配情况，它包含topologyId以及为该Topology分配的<ExecutorDetails,WorkerSlot>映射关系。
- Cluster对象定义了集群当前的状态信息，包括所有Supervisor信息以及当前所有Topology的分配信息等，它是任务调度器进行任务分配、调度的基础。

6.2.2　Clojure数据结构

Nimbus使用Clojure语言定义了一些共享数据结构以及Storm元数据，其中最主要的数据结构是nimbus-data,Storm元数据包括StormBase、Assignment和SupervisorInfo。

1. nimbus-data介绍

nimbus-data数据结构定义了很多公用数据，其代码如下所示：

```
1    (defn nimbus-data [conf inimbus]
2        (let [forced-scheduler (.getForcedScheduler inimbus)]
3           {:conf conf
4           :inimbus inimbus
5           :submitted-count (atom 0)
6           :storm-cluster-state (cluster/mk-storm-cluster-state conf)
7           :submit-lock (Object.)
8           :heartbeats-cache (atom {})
9           :downloaders (file-cache-map conf)
10          :uploaders (file-cache-map conf)
11          :uptime (uptime-computer)
12          :validator (new-instance (conf NIMBUS-TOPOLOGY-VALIDATOR))
13          :timer (mk-timer :kill-fn (fn [t]
14               (log-error t "Error when processing event")
15               (halt-process! 20 "Error when processing an event")
16               ))
17         :scheduler (mk-scheduler conf inimbus)
18         }))
```


- 第5行定义了当前已经提交的Topology的数目。
- 第6行定义了cluster-state对象，该对象可用于将数据存储到ZooKeeper中以及从ZooKeeper读取数据。
- 第9行和第10行分别定义了downloaders和uploaders缓存。当用户提交Topology的时候，系统会创建一个上传流并将其放入uploaders缓存中；而当Supervisor从Nimbus下载Topology的jar包时，系统则会创建一个下载流并将其放入downloaders缓存中。任何一种操作完成时，其所对应的上传或下载流就会被关闭，且流所传递的内容也会被从缓存中移除。
- 第11行定义了当前Nimbus的启动时间。
- 第12行定义了一个validator，它可用于对Topology进行检测验证。目前使用的是DefaultValidator，其validate操作实际上什么都没做。
- 第13~16行定义了一个计时器，并给出了当计时器处理失败时需要调用的方法。
- 最后一行则定义了Nimbus所使用的调度器。

2. StormBase

StormBase定义了Topology的基本信息，其代码如下：

```
(defrecord StormBase [storm-name launch-time-secs status num-workers component->executors])
```

它主要包括如下参数。

- storm-name：Topology名字。
- launch-time-secs：启动时间。
- status：Topology的当前状态。
- num-workers：Topology需要使用的Worker的数目。

- component->executors：保存了<component-id, parallelism>信息。

3. Assignment

Assignment定义了当前Topology的任务分配情况，其代码如下：

```
(defrecord Assignment [master-code-dir node->host executor->node+port executor->start-time-secs])
```

它主要包括如下参数。

- master-code-dir：Nimbus在本地保存该Topology信息的路径，其中主要含有三个文件——stormjar.jar、stormcode.ser以及stormconf.ser。
- node->host：定义了该Topology被分配到的<supervisor-id, hostname>集合。
- executor->node+port：定义了该Topology中Executor的分配情况，node对应于supervisor-id，port则是Supervisor的一个端口。
- executor->start-time-secs：定义了该Topology对应的Supervisor的启动时间。

4. SupervisorInfo

SupervisorInfo定义了Supervisor本身的信息，其代码如下：

```
(defrecord SupervisorInfo [time-secs hostname assignment-id used-ports meta scheduler-meta uptime-secs])
```

它主要包含如下参数。

- time-secs：最近一次更新SupervisorInfo数据的时间。
- hostname：Supervisor所在机器的主机名。
- assignment-id：目前等同于supervisor-id。
- used-ports：当前已被使用的端口列表。
- meta：Supervisor的元数据信息，目前主要用来记录Supervisor的所有端口列表。
- scheduler-meta：Supervisor的调度元数据信息，目前没有使用。
- uptime-secs：Supervisor截至上次心跳更新时的启动时间。

6.3 Nimbus 中的线程介绍

除主服务线程之外，Nimbus中还有一个计时器线程，它的主要作用有3个，具体如下所示。

- 调用mk-assignment方法启动新一轮的任务分配，调用do-cleanup方法清理Storm元数据。这两项操作会每隔NIMBUS-MONITOR-FREQ-SECS（默认值为10秒）执行一次。
- 调用clean-inbox方法清理Nimbus本地目录中Topology的jar包。该操作则每隔NIMBUS-CLEANUP-INBOX-FREQ-SECS（默认值是600秒）执行一次。
- 执行Topology的状态转移事件。这一事件则只有当Nimbus接收到对应的服务请求（如kill、rebalance、activate和deactivate）时才会被触发。

这里我们先来介绍会被周期性调用的3个方法：mk-assignments、do-cleanup和clean-inbox。而状态转移事件的执行过程，我们则留到下一节中单独讨论。

6.3.1 mk-assignments

mk-assignments方法主要负责对当前集群中所有Topology进行新一轮的任务调度。它一方面会检查已运行的Topology所占用的资源，判断它们是否有问题，是否需要重新分配；另一方面也会根据系统当前的可用资源，为新提交的Topology分配任务。然后，mk-assignments方法会将所有的分配信息保存或更新到ZooKeeper中，Supervisor会周期性地检查这些分配信息，并根据这些分配信息做相应的调度处理。该方法的代码如下所示：

```
1 (defnk mk-assignments [nimbus :scratch-topology-id nil]
2    (let [conf (:conf nimbus)
3         storm-cluster-state (:storm-cluster-state nimbus)
4         ^INimbus inimbus (:inimbus nimbus)
5         ;; read all the topologies
6         topology-ids (.active-storms storm-cluster-state)
7         topologies (into {} (for [tid topology-ids]
8                              {tid (read-topology-details nimbus tid)}))
9         topologies (Topologies. topologies)
10        ;; read all the assignments
11        assigned-topology-ids (.assignments storm-cluster-state nil)
12        existing-assignments (into {} (for [tid assigned-topology-ids]
13           (when (or (nil? scratch-topology-id) (not= tid scratch-topology-id))
14                 {tid (.assignment-info storm-cluster-state tid nil)})))
15        ;; make the new assignments for topologies
16        topology->executor->node+port (compute-new-topology->executor->node+port
17            nimbus
18            existing-assignments
19            topologies
20            scratch-topology-id)
21
22        now-secs (current-time-secs)
23        basic-supervisor-details-map (basic-supervisor-details-map storm-cluster-state)
24        ;; construct the final Assignments by adding start-times etc into it
25        new-assignments (into {} (for [[topology-id executor->node+port] topology->executor->node+port
26            :let [existing-assignment (get existing-assignments topology-id)
27                 all-nodes (->> executor->node+port vals (map first) set)
28                 node->host (->> all-nodes
29                     (mapcat (fn [node]
30                        (if-let [host (.getHostName inimbus basic-supervisor-details-map node)]
31                         [[node host]]
32                         )))
33                         (into {}))
34                 all-node->host (merge (:node->host existing-assignment) node->host)
35                 reassign-executors (changed-executors (:executor->node+port existing-assignment)
                     executor->node+port)
36                 start-times (merge (:executor->start-time-secs existing-assignment)
37                     (into {}
38                         (for [id reassign-executors]
39                         [id now-secs]
40                         )))]]
41                 {topology-id (Assignment.
42                     (master-stormdist-root conf topology-id)
```

```
43                  (select-keys all-node->host all-nodes)
44                  executor->node+port
45                  start-times)}))]
46
47    (doseq [[topology-id assignment] new-assignments
48        :let [existing-assignment (get existing-assignments topology-id)
49             topology-details (.getById topologies topology-id)]]
50        (if (= existing-assignment assignment)
51            (log-debug "Assignment for " topology-id " hasn't changed")
52            (do
53              (log-message "Setting new assignment for topology id " topology-id ": " (pr-str assignment))
54              (.set-assignment! storm-cluster-state topology-id assignment)
55             )))
56    (->> new-assignments
57        (map (fn [[topology-id assignment]]
58        (let [existing-assignment (get existing-assignments topology-id)]
59           [topology-id (map to-worker-slot (newly-added-slots existing-assignment assignment))]
60            )))
61        (into {})
62        (.assignSlots inimbus topologies))
63    ))
```

在mk-assignments方法中，主要涉及两个参数，具体如下所示。

- nimbus：nimbus-data对象。
- :scratch-topology-id：需要进行重新调度的Topology的id，也即storm-id。

下面我们来分析一下代码。

- 第2~4行分别从nimbus-data中获取conf、storm-cluster-state和inimbus对象并将其保存为临时变量。
- 第6行获取当前所有活跃的Topology的id集合。
- 第7~8行根据前面获取到的活跃topology-id集合，对每一个id调用read-topology-details方法，获取TopologyDetails信息并返回<storm-id, TopologyDetails>集合。
- 第9行利用前面返回的<storm-id, TopologyDetails>集合创建Topologies对象。
- 第11行获取所有已经分配资源的Topology的id集合。
- 第12~14行根据前面得到的已分配资源的topology-id集合，获取其中每个Toplogy的任务分配结果assignments，并返回<storm-id, Assignment>集合。注意，对于需要重新调度的Topology（由scratch-topology-id指定），这里将不会去获取它的assignments，其所有的slots也都会被视为可用资源。
- 第16~20行调用compute-new-topology->executor->node+port方法为所有的Topology计算新的调度，并返回topology->executor->node+port。我们会在后面对该方法进行介绍。
- 第22行获取当前系统时间（转化成秒）。
- 第23行调用basic-supervisor-details-map方法获取ZooKeeper中所有的SupervisorInfo信息，然后将其转换为<supervisor-id, SupervisorDetails>集合。
- 第25~45行对前面第16~20行返回的topology->executor->node+port中的每一项进行处理，

最终的返回值是新计算得到的<topology-id, Assignment>集合。

- 第26行根据topology-id从第12~14行返回的<storm-id, Assignment>集合中获取Topology的任务分配情况。
- 第27行根据executor->node+port信息提取其中所有的节点信息。
- 第28~33行根据第27行返回的node集合，尝试获取每一个节点的主机名信息，并返回一个<node, hostname>集合。
- 第34行会将从第26行获取的assignment中的<node, host>集合，跟从第28~33行获取的<node, hostname>集合进行合并，得到所有的<node, host>关系。如果存在相同的node，那么与其对应的主机名将采用<node, hostname>集合中的值，即以从第28 ~ 33行获取的信息为准。
- 第35行调用changed-executors方法，通过将第26行返回的assignment中的executor->node+port信息同新计算得到的executor->node+port信息进行对比，计算出所有被重新分配的Executor。
- 第36~40行通过将已经存在的assignment中的executor->start-time-secs信息与所有被重新分配的executor->start-time-secs信息（它们的start-time-secs为当前时间）进行合并，获得最新的所有<executor, start-time-secs>集合。
- 第41~45行创建新的Assignment对象。其参数分别是该Topology在Nimbus服务器上的root文件夹路径、<node, host>集合、新的executor->node+port映射关系集合以及新的executor->start-time-secs映射关系集合。
- 第47~55行对于前面返回的新计算的<topology-id, Assignment>集合中的每一项，比较其新调度（new-assignments）跟目前调度（existing-assignments）之间是否发生了变化。如果没有，就打印一条记录；如果有，则将该Topology在ZooKeeper中保存的调度结果更新为new-assignments。
- 第56~63行对于前面返回的new-assignments中的每一项，首先计算出新增加的slot（通过将new-assignments中的node+port减去existing-assignments中的node+port得到），再将其转化为worker-slot对象，返回的是<topology-id, WorkerSlot>集合，最后调用inimbus的assignSlots方法来分配slot（貌似现在这一步没什么用，这个assignSlots方法什么都没做）。

6.3.2 do-cleanup

该方法用于判断哪些Topology需要清理，并对需要清理的Topology进行相应的操作。do-cleanup会首先删除这些Topology保存在ZooKeeper中的心跳及错误信息，然后尝试清除Nimbus本地目录中的相关文件，并从Nimbus心跳缓存中移除对应的信息。do-cleanup方法的代码如下所示：

```
1 (defn do-cleanup [nimbus]
2    (let [storm-cluster-state (:storm-cluster-state nimbus)
3         conf (:conf nimbus)
4         submit-lock (:submit-lock nimbus)]
```

6

```
5   (let [to-cleanup-ids (locking submit-lock
6      (cleanup-storm-ids conf storm-cluster-state))]
7      (when-not (empty? to-cleanup-ids)
8          (doseq [id to-cleanup-ids]
9              (log-message "Cleaning up " id)
10             (.teardown-heartbeats! storm-cluster-state id)
11             (.teardown-topology-errors! storm-cluster-state id)
12             (try
13                 (rmr (master-stormdist-root conf id))
14             (catch Exception e (log-warn (.getMessage e))))
15             (swap! (:heartbeats-cache nimbus) dissoc id))
16         ))))
```

6.3.3 clean-inbox

该方法负责清理Nimbus的inbox文件夹（其路径是STORM-LOCAL-DIR/nimbus/inbox），清理的条件是从jar包最后一次被修改到当前时间的间隔超过了`NIMBUS-INBOX-JAR-EXPIRATION-SECS`的限定（默认是3600秒）。该方法的代码如下：

```
1 (defn clean-inbox [dir-location seconds]
2   "Deletes jar files in dir older than seconds."
3   (let [now (current-time-secs)
4         pred #(and (.isFile %) (file-older-than? now seconds %))
5         files (filter pred (file-seq (File. dir-location)))]
6         (doseq [f files]
7             (if (.delete f)
8                 (log-message "Cleaning inbox ... deleted: " (.getName f))
9                 ;; This should never happen
10                (log-error "Cleaning inbox ... error deleting: " (.getName f))
11                ))))
```

6.4　Topology 状态转移

Nimbus需要监视当前所有Topology的状态，并根据收到的Topology状态转移请求（如kill、rebalance、activate和deactivate等）完成相应的状态转换。在Nimbus中，我们定义了通用的状态转移方法，它们会根据传入的转移事件做相应的处理，这些方法包括`transition-name!`、`transition!`和`state-transitions`。

6.4.1 transition-name!

该方法会根据`topology-name`及对应的转移事件完成Topology的状态转换，相关代码如下：

```
1 defn transition-name! [nimbus storm-name event & args]
2     (let [storm-id (get-storm-id (:storm-cluster-state nimbus) storm-name)]
3         (when-not storm-id
4             (throw (NotAliveException. storm-name)))
5         (apply transition! nimbus storm-id event args)))
```

这个方法很简单，首先它通过调用get-storm-id方法，为传入的storm-name查找对应的storm-id，如果找到了，就调用transition!方法。所以它的作用实际上就是将基于storm-name的状态转换为基于storm-id的状态。

6.4.2 transition!

transition!方法会根据传入的转移事件，获取与当前Topology状态对应的状态转换函数，并执行该函数取得转换后的新状态。如果新状态不为空，则将其更新到ZooKeeper中。该方法的代码如下：

```
1 (defn transition!
2     ([nimbus storm-id event]
3        (transition! nimbus storm-id event false))
4     ([nimbus storm-id event error-on-no-transition?]
5         (locking (:submit-lock nimbus)
6             (let [system-events #{:startup}
7                [event & event-args] (if (keyword? event) [event] event)
8              status (topology-status nimbus storm-id)]
9              ;; handles the case where event was scheduled but topology has been removed
10             (if-not status
11                 (log-message "Cannot apply event " event " to " storm-id " because topology no
                        longer exists")
12                 (let [get-event (fn [m e]
13                         (if (contains? m e)
14                            (m e)
15                            (let [msg (str "No transition for event: " event
16                                ", status: " status,
17                                " storm-id: " storm-id)]
18                                (if error-on-no-transition?
19                                    (throw-runtime msg)
20                                    (do (when-not (contains? system-events event)
21                                        (log-message msg))
22                                    nil))
23                            )))
24                 transition (-> (state-transitions nimbus storm-id status)
25                     (get (:type status))
26                     (get-event event))
27                 transition (if (or (nil? transition)
28                     (keyword? transition))
29                     (fn [] transition)
30                     transition)
31                 new-status (apply transition event-args)
32                 new-status (if (keyword? new-status)
33                     {:type new-status}
34                     new-status)]
35             (when new-status
36                 (set-topology-status! nimbus storm-id new-status)))))
37        )))
```

- 第2~3行是一个重载方法，它会直接调用第4行的方法。默认的参数false表示在找不到对应的转换方式时将不抛出异常。

- 第5行尝试获取nimbus-data的submit-lock。为了提高Nimbus的处理性能，在storm所采取的处理模式中（后面介绍state-transition时我们会说明这一点），无论是Nimbus的主服务线程还是计时器线程都会调用transition方法。因此为了确保逻辑的正确性，必须在这里做同步。
- 第6行定义了一个system-events集合，它只包含:startup。
- 第7行判断event是否是keyword（以“:”开头）。如果是，会返回event的一个集合形式；如果不是，则照原样返回。其中，第一个参数绑定为event，即转移事件；剩下的参数则绑定为event-args，它是转移事件所对应的状态转换函数的参数。
- 第8行调用topology-status来获取Topology当前的运行状态。
- 在第10~11行中如果没有找到Topology的运行状态，即Topology已被移除，那么打印日志并结束处理，否则就继续执行下面的代码。
- 第12~23行定义了get-event方法，该方法会根据键从一个哈希表中查找对应的状态转换函数。如果没能找到，则将这一情况记录下来并根据error-on-no-transition的设置做相应的处理。
- 第24~26行首先调用state-transitions方法获取所有的状态跟状态间的转移事件（即一个哈希表，键是原状态，值是一个从转移事件到状态转移函数的哈希表），然后根据Topology的状态找到对应的从转移事件到状态转移函数的哈希表，最后调用get-event方法根据传入的转移事件获取状态转换函数。
- 第27~30行对前面获取的状态转移函数进行处理。如果该函数为空（也即不需要转移）或者该函数是一个关键字（直接返回转移后的新状态），那么就用一个函数对其进行封装，否则不做处理。
- 第31行直接调用前面处理过的状态转移函数，并根据传入的event-args参数，获取转移后的新状态。
- 第32~34行对得到的新状态进行判断。如果它是关键字（keyword），那么将它封装成一个哈希表，键是:type；否则就不做任何处理。
- 第35~36行判断新的状态是否为空，若不为空，则更新这个Topology在ZooKeeper中保存的状态。

6.4.3 state-transitions

state-transitions方法定义了一个状态转移矩阵，它的键集合包括:active、:inactive、:killed以及:rebalancing，它们表示当前Topology所处的全部起始状态，它的值定义了Topology处于由键指定的状态时，其状态变化需要遵循的从转移事件到对应状态转移函数的映射集合。其相关代码如下：

```
1 defn state-transitions [nimbus storm-id status]
2   {:active {:inactivate :inactive
3       :activate nil
```

```
4        :rebalance (rebalance-transition nimbus storm-id status)
5        :kill (kill-transition nimbus storm-id)
6        }
7    :inactive {:activate :active
8        :inactivate nil
9        :rebalance (rebalance-transition nimbus storm-id status)
10       :kill (kill-transition nimbus storm-id)
11       }
12   :killed {:startup (fn [] (delay-event nimbus
13       storm-id
14       (:kill-time-secs status)
15       :remove)
16       nil)
17       :kill (kill-transition nimbus storm-id)
18       :remove (fn []
19           (log-message "Killing topology: " storm-id)
20           (.remove-storm! (:storm-cluster-state nimbus)
21               storm-id)
22           nil)
23       }
24   :rebalancing {:startup (fn [] (delay-event nimbus
25       storm-id
26       (:delay-secs status)
27       :do-rebalance)
28       nil)
29       :kill (kill-transition nimbus storm-id)
30       :do-rebalance (fn []
31            (do-rebalance nimbus storm-id status)
32            (:old-status status))
33        }})
```

- 第2~6行表示当Topology的状态为:active时，该Topology正在运行。:inactivate事件会使Topology状态转移到:inactive，:activate事件不会改变Topology的状态，:rebalance事件会触发rebalance-transition函数，:kill事件会触发kill-transition函数。
- 第7~11行表示当Topology的状态为:inactive时，该Topology已经停止运行。:activate事件会使Topology状态转移到:active，:inactivate事件不会改变Topology的状态，:rebalance事件会触发rebalance-transition函数，:kill事件则会触发kill-transition函数。
- 第12~23行表示当Topology的状态为:killed时，该Topology已经被删除，但ZooKeeper中的相关数据依然没被删除。:startup事件会触发delay-event函数，此时Topology的状态并不会发生改变。:kill事件会触发kill-transition函数。:remove事件则会触发其对应的fn函数，用于将该Topology的信息从ZooKeeper中删除，这种情况下返回nil信号只是为了确保Storm不会在ZooKeeper中设置该Topology的状态。
- 第24~32行表示当Topology的状态为:rebalancing时，Storm正准备为该Topology重新分配资源。:startup事件会触发delay-event函数，此时Topology的状态并不会发生改变，:kill事件则触发kill-transition函数，:do-rebalance事件会触发do-rebalance函数，函数执行完成后会将Topology状态设置为进行rebalance操作之前的状态。

6

看到这里，估计很多人会觉得奇怪，为什么Storm中会有:killed和:rebalancing这两个状态呢。按照正常的逻辑，删除一个Topology后根本不需要再去关心它的状态，进行rebalance操作，然后返回之前的状态就可以了，因此这两个状态貌似有点画蛇添足。但实际上Storm这样做是有其用意的：为了提高Nimbus的处理速度，很多操作都被分为两步来执行，Nimbus的主服务线程往往只是将这些事件转发后便立即返回，具体的操作则都交由计时器线程来完成。下面我们来分析kill-transition和rebalance-transition方法，相信看完这两个方法大家就会明白Storm这样设计的用意了。

在分析之前，我们先看一下delay-event方法，这个方法在kill-transition和rebalance-transition中都用到了。

delay-event方法表示延迟一段时间后再处理转移事件，其参数包括nimbus-data、storm-id、延迟执行的时间及转移事件。它实际上就是通过简单调用schedule方法（这个方法在第4章介绍过）将创建的匿名函数加入到Nimbus计时器线程的优先级队列中。而这个匿名函数就是调用transition!方法来处理转移事件。该方法的代码如下：

```
1  (defn delay-event [nimbus storm-id delay-secs event]
2     (log-message "Delaying event " event " for " delay-secs " secs for " storm-id)
3     (schedule (:timer nimbus)
4        delay-secs
5        #(transition! nimbus storm-id event false)
6        ))
```

kill-transition方法定义了一个方法，其参数是kill-time，即做真正的kill操作之前需要等待的时间，其代码如下：

```
1 (defn kill-transition [nimbus storm-id]
2    (fn [kill-time]
3       (let [delay (if kill-time
4            kill-time
5            (get (read-storm-conf (:conf nimbus) storm-id)
6                 TOPOLOGY-MESSAGE-TIMEOUT-SECS))]
7      (delay-event nimbus
8          storm-id
9          delay
10         :remove)
11     {:type :killed
12     :kill-time-secs delay})
13   ))
```

- 第3~6行获取要等待的时间。如果传入的kill-time不为空，就采用kill-time，否则将使用TOPOLOGY-MESSAGE-TIMEOUT-SECS作为默认值。
- 第7~10行调用delay-event方法，event被设置为:remove，延迟时间被设置为前面计算的等待时间。
- 第11~12行返回方法执行的结果，其中Topology的状态被设置成了:killed。所以接下来，只有等到加入计时器线程的方法执行完后（在:killed状态下执行:remove转移事件），该

Topology在ZooKeeper中的数据才会被真正删除。由于这一步是在计时器线程中执行的，因此从总体上说，Nimbus的处理性能得到了提高。

与kill-transition类似，rebalance-transition也返回了一个方法，其代码如下：

```
1 (defn rebalance-transition [nimbus storm-id status]
2     (fn [time num-workers executor-overrides]
3          (let [delay (if time
4         time
5         (get (read-storm-conf (:conf nimbus) storm-id)
6             TOPOLOGY-MESSAGE-TIMEOUT-SECS))]
7         (delay-event nimbus
8             storm-id
9             delay
10            :do-rebalance)
11      {:type :rebalancing
12      :delay-secs delay
13      :old-status status
14      :num-workers num-workers
15      :executor-overrides executor-overrides
16      })))
```


返回的fn函数的参数有3个，具体如下所示。

- time：即在执行rebalance操作之前需要延迟的时间。
- num-workers：rebalance操作设置的新的num-workers。
- executor-overrrides：rebalance操作设置的新的从组件到Executor数目的哈希表。
- 同kill-transition类似，第3~6行也是获取等待时间。
- 第7~10行调用delay-event方法，event被设置为:do-rebalance，延迟时间被设置为前面计算得到的等待时间。
- 第11~15行返回执行结果。Topology的状态被更新为:rebalancing，表明rebalance操作正在进行中。等到加入计时器线程的方法执行后（在:rebalancing状态下执行:do-rebalance转移事件，也即执行do-rebalance方法），该Topology的rebalance操作才算真正完成。

最后简单介绍一下do-rebalance方法，其代码如下：

```
1 (defn do-rebalance [nimbus storm-id status]
2    (.update-storm! (:storm-cluster-state nimbus)
3        storm-id
4        (assoc-non-nil
5            {:component->executors (:executor-overrides status)}
6            :num-workers
7            (:num-workers status)))
8    (mk-assignments nimbus :scratch-topology-id storm-id))
```

该方法首先将rebalance设置的参数更新到ZooKeeper中，然后调用mk-assignments方法重新调度任务。这里将:scratch-topology-id设置为需要执行rebalance操作的Topology的id，这表明对于这个Topology，当前占有的所有资源在分配时都可以看作是可用的。这点在前面讲述

mk-assignments时曾经介绍过。

图6-2描述了与state-transition方法相对应的状态转移图。

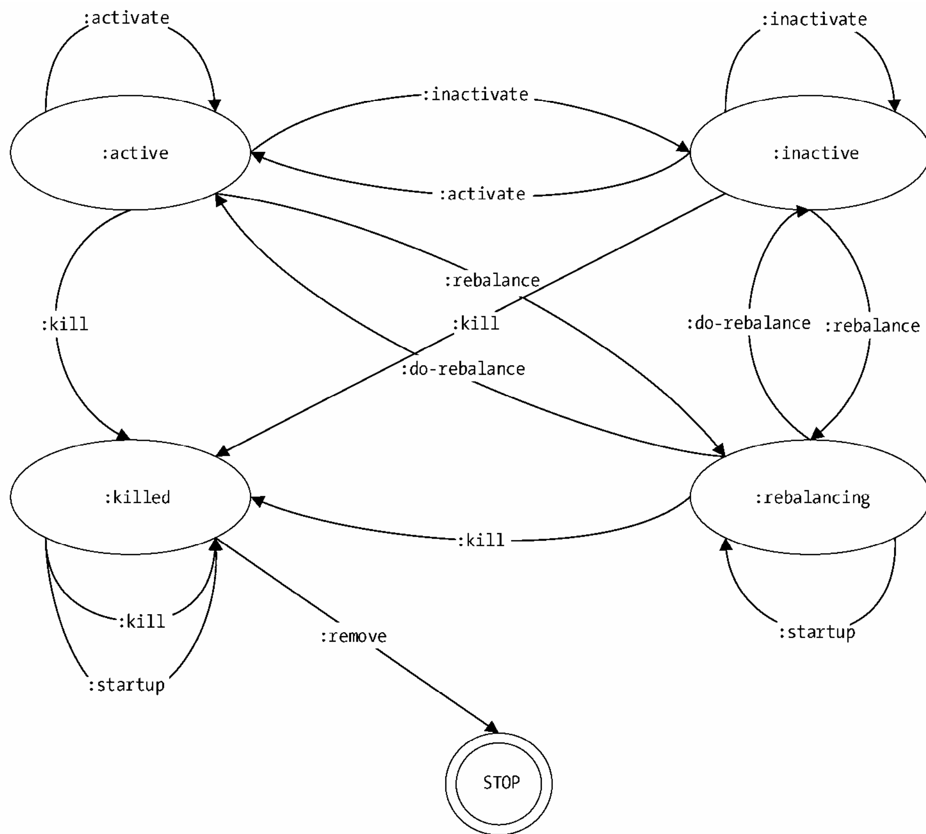


图6-2　Topology状态转移图

6.5　启动 Nimbus 服务

启动Nimbus服务主要涉及两个方法——launch-server!和service-handler，前者是Nimbus的启动入口，而后者则是真正定义处理逻辑的地方。

6.5.1　launch-server!

该方法是Nimbus服务的启动入口。它定义了核心的处理逻辑，构建起一个THsHaServer，并最终启动Nimbus服务，其代码如下：

```
1 (defn launch-server! [conf nimbus]
2   (validate-distributed-mode! conf)
```

```
3    (let [service-handler (service-handler conf nimbus)
4        options (-> (TNonblockingServerSocket. (int (conf NIMBUS-THRIFT-PORT)))
5           (THsHaServer$Args.)
6           (.workerThreads 64)
7           (.protocolFactory (TBinaryProtocol$Factory.))
8           (.processor (Nimbus$Processor. service-handler))
9           )
10       server (THsHaServer. options)]
11       (.addShutdownHook (Runtime/getRuntime) (Thread. (fn [] (.shutdown service-handler) (.stop
            server))))
12       (log-message "Starting Nimbus server...")
13       (.serve server)))
```

launch-server!的输入参数有以下两个。

- conf：配置项。
- nimbus：一个实现了INimbus接口的对象。

下面简要介绍各行代码的作用。

- 第2行验证当前系统是否处于分布式模式，因为只有处于分布式模式时才能够这样启动Nimbus 服务。
- 第3行定义了service-handler方法。顾名思义，它是Nimbus真正处理请求的地方，后面我们会介绍该方法。
- 第4~9行定义了用于启动Thrift ThsHaServer的参数，其中指定了服务器的端口号是NIMBUS-THRIFT-PORT，Worker线程池的线程数目是64，使用的协议是TBinaryProtocol，并以第3行定义的service-handler作为该服务器的处理器。
- 第10行使用前面定义的参数创建ThsHaServer。
- 第11行为当前JVM添加一个关闭的钩子，该钩子实际上是已被初始化但还没有开始执行的线程对象。当JVM将要停止时，这些钩子便开始执行，完成诸如停止service-handler和关闭ThsHaServer之类的清理工作。
- 第13行调用THsHaServer的serve方法来启动服务。至此，Nimbus 服务就算是成功启动了，接下来我们便可开始进行提交Topology等例行操作了。

6.5.2 service-handler

service-handler方法是Nimbus真正处理请求的地方，它首先定义了一些必要的数据结构，以及用于启动任务调度和数据清理的线程；其次，它还会返回一个实现了Nimbus$Iface接口、Shutdownable接口以及DaemonCommon接口的对象，Nimbus负责处理请求信息的逻辑都包含在这个对象中。

下面我们来看一个这个方法的具体实现：

```
1 (defserverfn service-handler [conf inimbus]
2   (.prepare inimbus conf (master-inimbus-dir conf))
3   (log-message "Starting Nimbus with conf " conf)
```

```
4    (let [nimbus (nimbus-data conf inimbus)]
5       (cleanup-corrupt-topologies! nimbus)
6       (doseq [storm-id (.active-storms (:storm-cluster-state nimbus))]
7           (transition! nimbus storm-id :startup))
8       (schedule-recurring (:timer nimbus)
9            0
10           (conf NIMBUS-MONITOR-FREQ-SECS)
11           (fn []
12               (when (conf NIMBUS-REASSIGN)
13                   (locking (:submit-lock nimbus)
14                       (mk-assignments nimbus)))
15           (do-cleanup nimbus)
16           ))
17     ;; Schedule Nimbus inbox cleaner
18     (schedule-recurring (:timer nimbus)
19        0
20        (conf NIMBUS-CLEANUP-INBOX-FREQ-SECS)
21        (fn []
22            (clean-inbox (inbox nimbus) (conf NIMBUS-INBOX-JAR-EXPIRATION-SECS))
23            ))
24     (reify Nimbus$Iface
25         ...... （详细实现后面我们会介绍）
26         Shutdownable
27         ......
28         DaemonCommon
29        (waiting? [this]
30            (timer-waiting? (:timer nimbus))))))
```

该方法涉及的参数有如下两个。

- conf：Nimbus使用的配置信息。
- inimbus：standalone-nimbus对象。

下面简要介绍该方法中各行代码的含义。

- 第2行调用inumbus的prepare方法，目前的实现中这个prepare方法为空。
- 第4行调用nimbus-data方法构建Nimbus数据结构，这个结构的具体内容已在前面介绍过。
- 第5行调用cleanup-corrupt-topologies!方法清除那些在ZooKeeper上还有元数据但在Nimbus本地目录中没有对应文件夹的Topology，将它们遗留在ZooKeeper中的记录彻底删除掉。
- 第6~7行对当前所有处于活跃状态的Topology调用transition!方法，设置Topology的状态。这里将转移事件设置为:start-up。
- 第8~23行使用nimbus-data中的计时器线程调度mk-assignments方法、do-cleanup方法及clean-inbox方法。这里要注意的是，mk-assignments方法只有在Storm配置项中NIMBUS-REASSIGN为true（默认是true）时才会执行，且在执行任务分配时需要获取:submit-lock锁对象，以避免任务分配和提交新Topology这两项操作发生冲突。
- 第24~30行返回一个实现了Nimbus$Iface、Shutdownable和DaemonCommon接口的对象，该对象会被用来处理Nimbus服务请求以及关闭Nimbus服务。

6.6　关闭 Nimbus 服务

关闭Nimbus服务要进行的操作主要包括杀掉计时器线程，释放ZooKeeper连接，以及清除nimbus-data中的上传、下载缓存。其实现方法如下：

```
1 (shutdown [this]
2     (log-message "Shutting down master")
3     (cancel-timer (:timer nimbus))
4     (.disconnect (:storm-cluster-state nimbus))
5     (.cleanup (:downloaders nimbus))
6     (.cleanup (:uploaders nimbus))
7     (log-message "Shut down master")
8 )
```

6.7　主要服务方法

下面我们对Nimbus的主要服务方法进行分析。它们包括Topology的提交操作、jar文件的上传与下载，以及UI信息、Storm配置项和Topology对象的获取等基本工作。所涉及的源码请参考storm\src\clj\backtype\storm\daemon\nimbus.clj。

6

6.7.1　submitTopology

该方法用来提交一个新的Topology，并为Topology创建topology-id，验证其结构，设置一些必要的元数据，最后调用mk-assignments方法为Topology分配任务。其代码如下：

```
1 (^void submitTopologyWithOpts
2     [this ^String storm-name ^String uploadedJarLocation ^String serializedConf ^StormTopology
          topology ^SubmitOptions submitOptions]
3     (try
4         (assert (not-nil? submitOptions))
5         (validate-topology-name! storm-name)
6         (check-storm-active! nimbus storm-name false)
7         (.validate ^backtype.storm.nimbus.ITopologyValidator (:validator nimbus)
8             storm-name
9             (from-json serializedConf)
10            topology)
11        (swap! (:submitted-count nimbus) inc)
12        (let [storm-id (str storm-name "-" @(:submitted-count nimbus) "-" (current-time-secs))
13            storm-conf (normalize-conf
14                conf
15                (-> serializedConf
16                    from-json
17                    (assoc STORM-ID storm-id)
18                    (assoc TOPOLOGY-NAME storm-name))
19                topology)
20                total-storm-conf (merge conf storm-conf)
21                topology (normalize-topology total-storm-conf topology)
22                topology (if (total-storm-conf TOPOLOGY-OPTIMIZE)
23                    (optimize-topology topology)
```

```
24                    topology)
25                storm-cluster-state (:storm-cluster-state nimbus)]
26            (system-topology! total-storm-conf topology) ;; this validates the structure of the
                topology
27            (log-message "Received topology submission for " storm-name " with conf " storm-conf)
28            ;; lock protects against multiple topologies being submitted at once and
29            ;; cleanup thread killing topology in b/w assignment and starting the topology
30            (locking (:submit-lock nimbus)
31                (setup-storm-code conf storm-id uploadedJarLocation storm-conf topology)
32                (.setup-heartbeats! storm-cluster-state storm-id)
33                (let [thrift-status->kw-status {TopologyInitialStatus/INACTIVE :inactive
34                    TopologyInitialStatus/ACTIVE :active}]
35                    (start-storm nimbus storm-name storm-id (thrift-status->kw-status
                        (.get_initial_status submitOptions))))
36                (mk-assignments nimbus)))
37            (catch Throwable e
38               (log-warn-error e "Topology submission exception. (topology name='"
                   storm-name "')")
39            (throw e))))
40
41 (^void submitTopology
42   [this ^String storm-name ^String uploadedJarLocation ^String serializedConf ^StormTopology
       topology]
43   (.submitTopologyWithOpts this storm-name uploadedJarLocation serializedConf topology
44           (SubmitOptions. TopologyInitialStatus/ACTIVE)))
```

这里主要有两个方法：submitTopologyWithOpts和submitTopology。后者是基于前者实现的，因此两个方法的执行过程类似，只不过submitTopology将默认的SubmitOptions设置为了ACTIVE。通常，我们提交Topology时，都是直接调用submitTopology方法。

下面我们分析一下代码。

- 第4行是对submitOptions参数进行验证，这个参数不能为空。
- 第5行调用validate-topology-name! 来确保storm-name中不含有非法字符，目前的非法字符集定义为 { "/" "." ";" "\"}。
- 第6行确保当前没有一个跟要提交的Topology同名（storm-name）的Topology正在运行。
- 第7~10行通过调用nimbus-data中的validator来对当前要提交的Topology进行验证，目前使用的DefaultValidator实际上什么都没做。
- 第11行用于更新nimbus-data中的：ubmitted-count参数（加1）。
- 第12行为这个要提交的Topology创建全局唯一的storm-id，也即topology-id，它的格式为<storm-name>-<当前的submitted-count>-<当前时间（转化成秒）>。
- 第13~19行通过调用normalize-conf方法获取要提交的Topology的Storm 配置，它首先对传入的serializedConf进行反序列化操作，然后加入storm-name和storm-id信息，最后再加入如下参数：TOPOLOGY-KRYO-DECORATORS、TOPOLOGY-KRYO-REGISTER、TOPOLOGY-ACKER-EXECUTORS、TOPOLOGY-MAX-TASK-PARALLELISM。这里如果用户的配置中指定了TOPOLOGY-KRYO-DECORATORS和TOPOLOGY-KRYO-REGISTER，Storm会优先使用用户的配置。

- 第20行将Storm 默认的配置项和前面获取的Storm配置项合并（其实主要是加入前面获取的Storm配置项中没有的但是Storm默认配置项中存在的配置项）。
- 第21行调用normalize-topology处理Topology，更新Topology中所有组件的TOPOLOGY-TASKS配置项，本章后面会介绍该方法。
- 第22~24行通过判断配置项中TOPOLOGY-OPTIMIZE是否为true来选择是否需要对Topology进行优化（实际上，Storm目前还没有这一优化的具体实现，即使配置为true，optimize-topology也是什么都没做）。
- 第25行获取nimbus-data中的:storm-cluster-state对象，并将其存储为临时变量供后面使用。
- 第26行通过调用system-topology!方法对Topology结构进行验证，我们会在后面介绍该方法。
- 第30行尝试获取nimbus-data的:submit-lock锁。
- 第31行调用setup-storm-code为该Topology创建对应的本地文件夹，复制.jar文件，并写入序列化后的Storm配置项和Topology信息。
- 第32行为该Topology在ZooKeeper中创建心跳路径，路径是/storm/workerbeats/topology-id。
- 第33~34行定义了一个从thrift-status到keyword-status的哈希表，这个哈希表会被用来将传入的submitOptions中的thrift-status转化为对应的keyword-status。
- 第35行调用start-storm方法设置stormBase，它在ZooKeeper中的路径是/storm/storms/<topology-id>，stormBase的信息将作为该路径所对应的值存储。
- 第36行调用mk-assignments为新提交的Topology分配资源，这个方法我们已经介绍过。

6.7.2 kill、rebalance、activate、deactivate 方法

接下来，我们分析kill、rebalance、activate和deactivate方法的实现。将这几个方法放在一起分析，主要是因为它们都依赖于前面介绍过的transition-name!函数。相关代码如下：

```
1 (^void killTopology [this ^String name]
2     (.killTopologyWithOpts this name (KillOptions.)))
3
4 (^void killTopologyWithOpts [this ^String storm-name ^KillOptions options]
5     (check-storm-active! nimbus storm-name true)
6     (let [wait-amt (if (.is_set_wait_secs options)
7               (.get_wait_secs options)
8               )]
9     (transition-name! nimbus storm-name [:kill wait-amt] true)
10    ))
11
12 (^void rebalance [this ^String storm-name ^RebalanceOptions options]
13     (check-storm-active! nimbus storm-name true)
14     (let [wait-amt (if (.is_set_wait_secs options)
15                       (.get_wait_secs options))
16               num-workers (if (.is_set_num_workers options)
17                             (.get_num_workers options))
18               executor-overrides (if (.is_set_num_executors options)
19                                    (.get_num_executors options)
```

```
20                                   {})]
21        (doseq [[c num-executors] executor-overrides]
22            (when (<= num-executors 0)
23                  (throw (InvalidTopologyException. "Number of executors must be greater
                       than 0"))
24           ))
25        (transition-name! nimbus storm-name [:rebalance wait-amt num-workers executor-overrides]
             true)
26        ))
27
28   (activate [this storm-name]
29       (transition-name! nimbus storm-name :activate true)
30       )
31
32   (deactivate [this storm-name]
33       (transition-name! nimbus storm-name :inactivate true))
```

从这段代码可以看到这些方法的实现基本上是类似的，不同的只是kill和rebalance多了一些必要的参数而已。

- 第5行确保提交杀掉Topology命令的时候这个Topology还是在运行着的。
- 第6~7行获取要延迟多少秒再杀掉这个Topology。这个时间是用户可以指定的，如果没有指定，将采用Topology的消息超时时间作为其默认值。
- 第13行跟第5行做了同样的工作。
- 第14~20行是获取需要的一些参数。
- 第21~24行用于验证用户的设定是否符合要求。

6.7.3　文件上传与下载

Nimbus作为一个服务器，一方面需要接收用户提交的Topology jar包，另一方面还要向Supervisor下达任务分配的jar包，所以上传和下载文件的功能是必需的。

1. 文件上传

文件上传的功能是通过beginFileUpload、uploadChunk和finishFileUpload这3个方法实现的，相关代码如下：

```
1 (beginFileUpload [this]
2     let [fileloc (str (inbox nimbus) "/stormjar-" (uuid) ".jar")]
3       (.put (:uploaders nimbus)
4       fileloc
5       (Channels/newChannel (FileOutputStream. fileloc)))
6       (log-message "Uploading file from client to " fileloc)
7     fileloc
8     ))
9
10 (^void uploadChunk [this ^String location ^ByteBuffer chunk]
11     (let [uploaders (:uploaders nimbus)
12       ^WritableByteChannel channel (.get uploaders location)]
13        (when-not channel
```

```
14          (throw (RuntimeException.
15              "File for that location does not exist (or timed out)")))
16        (.write channel chunk)
17     (.put uploaders location channel)
18 ))
19
20 (^void finishFileUpload [this ^String location]
21      (let [uploaders (:uploaders nimbus)
22        ^WritableByteChannel channel (.get uploaders location)]
23         (when-not channel
24            (throw (RuntimeException.
25                "File for that location does not exist (or timed out)")))
26         (.close channel)
27         (log-message "Finished uploading file from client: " location)
28         (.remove uploaders location)
29 ))
```

- beginFileUpload方法首先会创建一个全局唯一的文件名，然后创建一个新的WriteByteChannel，并将此关系保存到nimbus-data的:uploaders中，然后返回文件路径。
- uploadChunk方法首先根据文件路径从nimbus-data的:uploaders中找到对应的WriteByteChannel，然后把ByteBuffer中的数据写入到该channel中。
- finishFileUpload方法会按照文件路径从nimbus-data的:uploaders中找到对应的channel，将其关闭并将此文件路径从:uploaders中移除。

6

2. 文件下载

文件下载是由beginFileDownload和downloadChunk这两个方法实现的，相关代码如下：

```
1 (^String beginFileDownload [this ^String file]
2     (let [is (BufferFileInputStream. file)
3          id (uuid)]
4        (.put (:downloaders nimbus) id is)
5        id
6     ))
7
8 (^ByteBuffer downloadChunk [this ^String id]
9     (let [downloaders (:downloaders nimbus)
10         ^BufferFileInputStream is (.get downloaders id)]
11         (when-not is
12              (throw (RuntimeException.
13                 "Could not find input stream for that id")))
14     (let [ret (.read is)]
15         (.put downloaders id is)
16         (when (empty? ret)
17              (.remove downloaders id))
18         (ByteBuffer/wrap ret)
19 )))
```

- beginFileDownload方法会首先打开要下载的文件，创建一个BufferFileInputStream并赋予该流一个唯一的id，然后将此关系保存到nimbus-data的:downloaders中，返回这个唯一的id。

- downloadChunk方法首先根据id从nimbus-data的:downloaders中获取对应的BufferFile-InputStream，然后从该流中读取数据。如果读不到数据，则表明数据已经读完，此时会将该文件从:downloaders中移除。

6.7.4 获取UI所需的信息

由于Nimbus服务器本身记录了当前集群的任务分配和调度信息，这些信息需要通过UI展示给用户，所以Nimbus$Iface中还定义了获取这些信息的接口getClusterInfo和getTopologyInfo，下面简要介绍这两个方法。

1. etClusterInfo

该方法用来获取当前集群的统计信息，主要包括系统资源的占用情况、Nimbus服务运行了多长时间，以及当前系统中所有Topology的运行统计等。该方法的代码如下：

```
1 (^ClusterSummary getClusterInfo [this]
2   (let [storm-cluster-state (:storm-cluster-state nimbus)
3        supervisor-infos (all-supervisor-info storm-cluster-state)
4        ;; TODO: need to get the port info about supervisors...
5        ;; in standalone just look at metadata, otherwise just say N/A?
6        supervisor-summaries (dofor [[id info] supervisor-infos]
7            (let [ports (set (:meta info)) ;;only true for standalone]
8            (SupervisorSummary. (:hostname info)
9                                (:uptime-secs info)
10                               (count ports)
11                               (count (:used-ports info))
12                               id )
13                                       ))
14         nimbus-uptime ((:uptime nimbus))
15         bases (topology-bases storm-cluster-state)
16         topology-summaries (dofor [[id base] bases]
17                                 (let [assignment (.assignment-info storm-cluster-state id nil)]
18                                      (TopologySummary. id
19                                           (:storm-name base)
20                                           (->> (:executor->node+port assignment)
21                                                keys
22                                                (mapcat executor-id->tasks)
23                                                count)
24                                           (->> (:executor->node+port assignment)
25                                                 keys
26                                                count)
27                                           (->> (:executor->node+port assignment)
28                                                 vals
29                                                 set
30                                                 count)
31                                           (time-delta (:launch-time-secs base))
32                                           (extract-status-str base))
33                                      ))]
34      (ClusterSummary. supervisor-summaries
35          nimbus-uptime
36          topology-summaries)
37      ))
```

- 第3行调用all-supervisor-info获取<supervisor-id, SupervisorInfo> 信息。
- 在第6~13行中，对于每一个<supervisor-id, SupervisorInfo>信息，首先从SupervisorInfo的元数据中获取端口号信息。接下来，构造SupervisorSummary对象，其参数分别为主机名、启动时间、所有可以使用的端口的数目、使用的端口号的数目以及supervisor-id。最后，返回一个SupervisorSummary集合。
- 第14行获取Nimbus迄今为止的运行时间。
- 第15行调用topology-bases获取所有Topology的StormBase信息，并返回一个<topology-id, StormBase>集合。
- 第16~33行对前面获取到的<topology-id, StormBase>中的每一项，首先根据topology-id获取其任务分配信息，然后构建TopologySummary对象，其参数依次为topology-id、storm-name、所有的Task数目、所有的Executor数目、所有被占用的slot数目、到目前为止的启动时间以及Topology的当前状态等。最终返回的将是一个TopologySummary集合。
- 第34~36行利用前面获取的SupervisorSummary集合、Nimbus的启动时间以及TopologySummary集合，创建ClusterSummary对象并返回。

2. getTopologyInfo

该方法用来获取一个Topology的详细运行情况，包括其topology-id、topology-name、从启动到当前为止的运行时间、所有Executor的统计信息、当前运行状态以及运行过程中发生的错误信息，相关代码如下：

```
1 (^TopologyInfo getTopologyInfo [this ^String storm-id]
2    (let [storm-cluster-state (:storm-cluster-state nimbus)
3         task->component (storm-task-info (read-storm-topology conf storm-id) (read-
              storm-conf conf storm-id))
4         base (.storm-base storm-cluster-state storm-id nil)
5         assignment (.assignment-info storm-cluster-state storm-id nil)
6           beats (.executor-beats storm-cluster-state storm-id (:executor->node+port assignment))
7         all-components (-> task->component reverse-map keys)
8         errors (->> all-components
9             (map (fn [c] [c (get-errors storm-cluster-state storm-id c)]))
10            (into {}))
11        executor-summaries (dofor [[executor [node port]] (:executor->node+port assignment)]
12            (let [host (-> assignment :node->host (get node))
13                 heartbeat (get beats executor)
14                 stats (:stats heartbeat)
15                 stats (if stats
16                      (stats/thriftify-executor-stats stats))]
17                 (doto
18                      (ExecutorSummary. (thriftify-executor-id executor)
19                           (-> executor first task->component)
20                                       host
21                                       port
22                                       (nil-to-zero (:uptime heartbeat)))
23                                  (.set_stats stats))
24                             ))
25              ]
```

```
26          (TopologyInfo. storm-id
27              (:storm-name base)
28              (time-delta (:launch-time-secs base))
29              executor-summaries
30              (extract-status-str base)
31              errors
32              )
33         ))
```

- 第3行调用storm-task-info方法获取<task-id, component-id>集合。
- 第4行根据storm-id获取该Topology的StormBase信息。
- 第5行根据storm-id获取该Topology的Assignment信息。
- 第6行根据storm-id和Assignment中的:executor->node+port信息获取所有Executor的心跳信息，这里返回的是一个<executor, {:time-secs, :uptime, :stats}>集合，其中:time-secs代表Executor对应的Worker最后一次心跳的更新时间，:uptime代表到这个最后一次更新心跳时的运行时间，:stats则代表Worker心跳中记录的该Executor的运行统计信息。
- 第7行从第3行得到的集合中获取所有组件信息。
- 第8~10行从ZooKeeper中获取每一个组件产生的错误信息。
- 第11~24行对Assignment中:executor->node+port的每一项分别进行一系列操作。首先从Assignment中的:node->host信息获取主机信息，然后从第6行返回的结果中根据Executor获取心跳信息（也即{:time-secs, :uptime, :stats}），接下来从心跳信息中获取:stats信息。如果运行统计不为空，调用stats.clj中的thriftify-executor-stats方法将其转换为ExecutorStats对象。最后构造ExecutorSummary对象，并设置其运行统计为构造出的ExecutorStats对象，返回一个ExecutorSummary集合。
- 第26~32行创建TopologyInfo对象并返回，其参数有storm-id、storm-name、迄今为止的启动时间、前面返回的ExecutorSummary集合、当前Topology状态以及所有出错信息。

6.7.5　获 取 Topology

Nimbus提供了两个获取Topology的方法，其中getUserTopology是获取用户提交的Topology，返回的StormTopology对象中不含有任何系统组件（acker-bolt、metric-bolt或者system-bolt）；getTopology方法获取真正在集群上运行的Topology，返回的StormTopology对象中包含所有系统组件。下面简要介绍这两个方法。

- getUserTopology方法直接调用read-storm-topology方法，根据topology-id获取存储在Nimbus本地目录中的StormTopology对象：

```
1 (^StormTopology getUserTopology [this ^String id]
2     (read-storm-topology conf id))
```

- getTopology方法首先根据topology-id从Nimbus本地目录中读取其对应的StormTopology对象和它使用的Storm配置项，然后调用system-topology!方法为该StormTopology对象添

加系统组件，包括acker-bolt、metric-bolt以及system-bolt，最后返回添加完这些组件的StormTopology对象：

```
1 (^StormTopology getTopology [this ^String id]
2     (system-topology! (read-storm-conf conf id) (read-storm-topology conf id)))
```

6.7.6 获取Storm配置项

Nimbus提供了获取Nimbus使用的Storm配置项的方法getNimbusConf和获取某个Topology使用的Storm配置项的方法getTopologyConf，下面简要介绍这两个方法。

- getNimbusConf方法直接返回JSON序列化后的nimbus-data中保存的Nimbus使用的Storm配置项：

```
1 (^String getNimbusConf [this]
2     (to-json (:conf nimbus)))
```

- getTopologyConf方法调用read-storm-conf方法从Nimbus本地目录中读取该Topology使用的Storm配置项，返回JSON序列化该对象后的字符串：

```
1 (^String getTopologyConf [this ^String id]
2     (to-json (read-storm-conf conf id)))
```


6.8 主要辅助方法

在介绍Nimbus的服务方法时，我们提到了一些很重要的辅助方法，这里简单介绍这些方法。

6.8.1 system-topology!

该方法主要用来验证用户提交的Topology，同时为用户提交的Topology添加一些系统组件和流，例如Acker、运行统计组件和系统组件等。该方法的代码如下：

```
1 (defn system-topology! [storm-conf ^StormTopology topology]
2     (validate-basic! topology)
3     (let [ret (.deepCopy topology)]
4          (add-acker! storm-conf ret)
5          (add-metric-components! storm-conf ret)
6          (add-system-components! storm-conf ret)
7          (add-metric-streams! ret)
8          (add-system-streams! ret)
9          (validate-structure! ret)
10         ret
11     ))
```

- 第2行调用validate-basic!方法验证用户提交的Topology是否符合要求。
- 第3行复制用户提交的Topology，并将其保存到ret变量中。

- 第4行调用add-acker!为ret添加acker-bolt。
- 第5行调用add-metric-components!为ret添加metric-bolt。
- 第6行调用add-system-components!为ret添加system-bolt。
- 第7行调用add-metric-streams!为ret中的所有组件添加统计流。
- 第8行调用add-system-streams!为ret中的所有组件添加系统流。
- 第9行调用validate-structure!方法验证添加完bolt和流后的Topology的正确性。
- 第10行返回新构造的ret对象。

下面我们会依次介绍其中涉及的一些方法：

1. validate-basic!

该方法负责验证用户创建的Topology是否符合要求，其代码如下：

```
1 (defn validate-basic! [^StormTopology topology]
2     (validate-ids! topology)
3     (doseq [f thrift/SPOUT-FIELDS
4         obj (->> f (.getFieldValue topology) vals)]
5         (if-not (empty? (-> obj .get_common .get_inputs))
6         (throw (InvalidTopologyException. "May not declare inputs for a spout"))))
7     (doseq [[comp-id comp] (all-components topology)
8         :let [conf (component-conf comp)
9         p (-> comp .get_common thrift/parallelism-hint)]]
10    (when (and (> (conf TOPOLOGY-TASKS) 0)
11         p
12         (<= p 0))
13      (throw (InvalidTopologyException. "Number of executors must be greater than 0 when number of
        tasks is greater than 0"))
14    )))
```

- 第2行调用validate-ids!方法确保Topology中：
  - 没有重复的组件id；
  - 组件id不是system id（以__开头）；
  - 流id不是system id。
- 第3~6行确保Spout没有输入。
- 第7~14行对Topology中的每一个组件进行判断，确保当TOPOLOGY-TASKS大于0时该组件设置的并行度一定也大于0。

2. add-acker!

Storm中采用Ack机制来追踪发送出去的消息是否被成功处理。关于这个机制后面专门有一章讲解。Storm默认提供了Acker Bolt的实现，所以用户不需要去关心这方面的内容。add-acker!方法负责为用户的Topology添加acker-bolt，并将加入的acker与用户组件关联起来。该方法的代码如下：

```
1 (defn add-acker! [storm-conf ^StormTopology ret]
2     (let [num-executors (storm-conf TOPOLOGY-ACKER-EXECUTORS)
3             acker-bolt (thrift/mk-bolt-spec* (acker-inputs ret)
4                             (new backtype.storm.daemon.acker)
5                             {ACKER-ACK-STREAM-ID (thrift/direct-output-fields ["id"])
```

```
6                               ACKER-FAIL-STREAM-ID (thrift/direct-output-fields ["id"])
7                               }
8                               :p num-executors
9                               :conf {TOPOLOGY-TASKS num-executors
10                                      TOPOLOGY-TICK-TUPLE-FREQ-SECS (storm-conf TOPOLOGY-MESSAGE-
                                            TIMEOUT-SECS)})]
11          (dofor [[_ bolt] (.get_bolts ret)
12              :let [common (.get_common bolt)]]
13              (do
14              (.put_to_streams common ACKER-ACK-STREAM-ID (thrift/output-fields ["id"
                    "ack-val"]))
15              (.put_to_streams common ACKER-FAIL-STREAM-ID (thrift/output-fields ["id"]))
16              ))
17       (dofor [[_ spout] (.get_spouts ret)
18                  :let [common (.get_common spout)
19                           spout-conf (merge
20                                          (component-conf spout)
21                                          {TOPOLOGY-TICK-TUPLE-FREQ-SECS (storm-conf
                                              TOPOLOGY-MESSAGE-TIMEOUT-SECS)})]]
22          (do
23          ;; this set up tick tuples to cause timeouts to be triggered
24           (.set_json_conf common (to-json spout-conf))
25           (.put_to_streams common ACKER-INIT-STREAM-ID (thrift/output-fields ["id" "init-val"
                  "spout-task"]))
26           (.put_to_inputs common
27               (GlobalStreamId. ACKER-COMPONENT-ID ACKER-ACK-STREAM-ID)
28               (thrift/mk-direct-grouping))
29           (.put_to_inputs common
30               (GlobalStreamId. ACKER-COMPONENT-ID ACKER-FAIL-STREAM-ID)
31               (thrift/mk-direct-grouping))
32           ))
33     (.put_to_bolts ret "__acker" acker-bolt)
34     ))
```

- ❑ 第2行获取用户设置的Acker Bolt节点数目。
- ❑ 第3~10行创建并行度为Acker Bolt数目的`acker-bolt`。
  - ■ 第3行调用`acker-inputs`方法获取acker的所有输入流，它首先从Topology的bolts和spouts分别获取所有的bolt的id列表和spout的id列表，然后对于每一个spout的id获取`{[id, ACKER-INIT-STREAM-ID] ["id"]}`,对于每一个bolt的id获取`{[id, ACKER-ACK-STREAM-ID] ["id"]}`和`{[id, ACKER-FAIL-STREAM-ID] ["id"]}`，最后将这两部分结果合并作为Acker Bolt的输入，它的结构是`<[component-id, streamtid], Grouping-Fields>`，所以这里`[id, ACKER-XXX-STREAM-ID]`中的id即为`component-id`，`ACKER-XXX-STREAM-ID`是`stream-id`，而`["id"]`表示`acker-bolt`从这些流接收数据是按照字段id进行分组的。
  - ■ 第4行创建新的`backtype.storm.daemon.acker`对象。
  - ■ 第5~7行创建`acker-bolt`的输出，它有两个输出流——`ACKER-ACK-STREAM-ID`和`ACKER-FAIL-STREAM-ID`，这两个流的输出都是只有一个字段id，并且这两个流都为直接流。
  - ■ 第8行设置`acker-bolt`的并行度。

6

    - 第9~10行定义创建acker-bolt的配置参数，它包含两项——TOPOLOGY-TASKS和TOPOLOGY-TICK-TUPLE-FREQ-SECS，前者设置为acker-bolt的并行度，后者设置为从storm-conf中获取的TOPOLOGY-MESSAGE-TIMEOUT-SECS。
- 第11~16行对于Topology中的每一个bolt，为其中的ComponentCommon对象的streams变量添加两个流：第一个是ACKER-ACK-STREAM-ID，设置它的字段名列表是["id","ack-val"]，设置它的模式为非直接模式；第二个是ACKER-FAIL-STREAM-ID，设置它的字段名列表为是["id"]，设置它的模式是非直接模式。
- 第17~32行对于Topology中的每一个Spout，为其中的ComponentCommon对象做以下操作。
    - 更新配置中的TOPOLOGY-TICK-TUPLE-FREQ-SECS为storm-conf中的TOPOLOGY-MESSAGE-TIMEOUT-SECS。
    - 在streams中添加ACKER-INIT-STREAM-ID，设置它的字段名列表为["id" "init-val" "spout-task"]，设置它的模式为非直接流。
    - 在输入流中添加直接流（__ack_ack）。
    - 在输入流中添加直接流（__acker_fail）。
- 第33行将创建的acker-bolt加入到Topology中，设置它的组件ID为__acker。

3. add-metric-components!

该方法为用户的Topology添加metric-bolt，其代码如下：

```
1 (defn add-metric-components! [storm-conf ^StormTopology topology]
2   (doseq [[comp-id bolt-spec] (metrics-consumer-bolt-specs storm-conf topology)]
3       (.put_to_bolts topology comp-id bolt-spec)))
```

第2~3行调用metrics-consumer-bolt-specs方法获取一个<component-id, metric-bolt>集合，将其中的每一个都加入到Topology的bolts集合中。关于metric-bolt的详细信息，后面有一章专门介绍。

4. add-system-components!

该方法为用户的Topology添加system-bolt，相关代码如下：

```
1 (defn add-system-components! [conf ^StormTopology topology]
2    (let [system-bolt-spec (thrift/mk-bolt-spec*
3        {}
4        (SystemBolt.)
5        {SYSTEM-TICK-STREAM-ID (thrift/output-fields ["rate_secs"])
6           METRICS-TICK-STREAM-ID (thrift/output-fields ["interval"])}
7        :p 0
8        :conf {TOPOLOGY-TASKS 0})]
9    (.put_to_bolts topology SYSTEM-COMPONENT-ID system-bolt-spec)))
```

- 第2~8行调用mk-bolt-spec*创建一个新的SystemBolt，这个Bolt没有输入流，输出流有两个，一个是SYSTEM-TICK-STREAM-ID，声明的字段是["rate_secs"]，非直接模式；另一个流是METRICS-TICK-STREAM-ID，声明的字段是["interval"]，也是非直接模式。并行度设置为0，配置设置为{TOPOLOGY-TASKS 0}。

❑ 第9行将新创建的System Bolt添加到Topology的Bolt列表中。

5. add-metric-streams!

该方法为Topology中每一个组件的ComponentCommon对象添加统计流：stream-id是METRICS-STREAM-ID，声明的字段是["task-info" "data-points"]，设置为非直接模式，相关代码如下：

```
1 (defn add-metric-streams! [^StormTopology topology]
2    (doseq [[_ component] (all-components topology)
3         :let [common (.get_common component)]]
4    (.put_to_streams common METRICS-STREAM-ID
5         (thrift/output-fields ["task-info" "data-points"]))))
```

6. add-system-streams!

该方法为Topology中每一个组件的ComponentCommon对象添加系统流：stream-id是SYSTEM-STREAM-ID，声明的字段是["event"]，设置为非直接模式，相关代码如下：

```
1 (defn add-system-streams! [^StormTopology topology]
2    (doseq [[_ component] (all-components topology)
3               :let [common (.get_common component)]]
4    (.put_to_streams common SYSTEM-STREAM-ID (thrift/output-fields ["event"]))))
```

6

7. validate-structure!

该方法验证添加完acker-bolt、metric-bolt以及system-bolt之后的StormTopology是否符合要求，相关代码如下所示：

```
1 (defn validate-structure! [^StormTopology topology]
2    ;; validate all the component subscribe from component+stream which actually exists in
        the topology
3    ;; and if it is a fields grouping, validate the corresponding field exists
4    (let [all-components (all-components topology)]
5       (doseq [[id comp] all-components
6                 :let [inputs (.. comp get_common get_inputs)]]
7          (doseq [[global-stream-id grouping] inputs
8                 :let [source-component-id (.get_componentId global-stream-id)
9                          source-stream-id    (.get_streamId global-stream-id)]]
10            (if-not (contains? all-components source-component-id)
11                (throw (InvalidTopologyException. (str "Component: [" id "] subscribes from
                    non-existent component [" source-component-id "]")))
12                    (let [source-streams (-> all-components (get source-component-id)
                       .get_common .get_streams)]
13           (if-not (contains? source-streams source-stream-id)
14              (throw (InvalidTopologyException. (str "Component: [" id "] subscribes from
                   non-existent stream: [" source-stream-id "] of component [" source-component-id
                       "]")))
15              (if (= :fields (thrift/grouping-type grouping))
16                  (let [grouping-fields (set (.get_fields grouping))
17                           source-stream-fields (-> source-streams (get source-stream-id)
                              .get_output_fields set)
18                           diff-fields (set/difference grouping-fields source-stream-fields)]
19                  (when-not (empty? diff-fields)
```

```
20                       (throw (InvalidTopologyException. (str "Component: [" id "] subscribes
                              from stream: [" source-stream-id "] of component [" source-component-id
                                  "] with non-existent fields: " diff-fields))))))))))))
```

- ❑ 第4行获取Topology中的所有组件，返回的是一个<component-id，组件>集合。
- ❑ 第5~20行对获取到的<component-id，组件>集合中的每一项进行处理。
  - ■ 第6行获取组件的所有输入。
  - ■ 第7~20行对输入中的每一项，获取它的source-component-id以及source-stream-id。首先判断source-component-id是否在组件列表中，如果不在，就抛出InvalidTopologyException异常。如在则继续判断source-stream-id是否在source-component-id对应的组件声明的流列表中，如果不在也会抛出InvalidTopologyException异常；如在则继续检查该流的分组方式是否是域分组（Fields Grouping），如果是，判断用来进行分组的域是否都在source-stream-id对应的流中存在，如果不存在也会抛出InvalidTopologyException。

如果所有的component检查完毕后都没有任何异常抛出，则说明该StormTopology是有效的。

6.8.2　normalize-topology

该方法主要用于计算提交的Topology中每个组件的并行度并更新该组件的TOPOLOGY-TASKS配置项，它的定义如下：

```
1 (defn normalize-topology [storm-conf ^StormTopology topology]
2    (let [ret (.deepCopy topology)]
3         (doseq [[_ component] (all-components ret)]
4             (.set_json_conf
5                 (.get_common component)
6                 (->> {TOPOLOGY-TASKS (component-parallelism storm-conf component)}
7                     (merge (component-conf component))
8                     to-json )))
9    ret ))
```

在上述代码中，输入参数storm-conf表示该Topology使用的配置项，topology是一个StormTopology对象，它包含了组成要提交的Topology的所有组件。下面简述各行代码的作用。

- ❑ 第2行调用StormTopology的deepCopy方法获取一个深度拷贝，这样做是因为Clojure是一个函数式编程语言，如果参数是一个对象，一般是不会修改该对象本身的，而是先复制一份，在副本上做修改，最后返回修改后的对象。
- ❑ 第3~8行首先调用all-components方法获取到该Topology中的所有组件信息，然后遍历每一个组件，更新组件配置中的TOPOLOGY-TASKS信息。TOPOLOGY-TASKS的计算是通过调用方法component-parallelism完成的。

下面简单介绍一下component-parallelism方法，它是用来计算组件并行度的，相关代码如下：

```
1 (defn- component-parallelism [storm-conf component]
2    (let [storm-conf (merge storm-conf (component-conf component))
3          num-tasks (or (storm-conf TOPOLOGY-TASKS) (num-start-executors component))
4          max-parallelism (storm-conf TOPOLOGY-MAX-TASK-PARALLELISM)
```

```
5         ]
6     (if max-parallelism
7         (min max-parallelism num-tasks)
8         num-tasks)))
```

- 第2行用于获取配置信息，该配置信息是将Topology的配置信息和组件的配置信息合并得到的，如果二者有相同的配置项，优先使用组件自己的配置项。
- 第3行计算num-tasks，也即该组件的并行度。如果第2行获取到的配置项中设置了TOPOLOGY-TASKS信息，就采用该设置作为num-tasks，否则通过调用num-start-executors方法获取用户为该组件设置的并行度作为num-tasks。
- 第4行从第2行获取到的配置信息中尝试获取配置项TOPOLOGY-MAX-TASK-PARALLELISM，将结果保存在max-parallelism中。
- 第6~8行判断第4行是否获取到配置项TOPOLOGY-MAX-TASK-PARALLELISM，如果获取到，就返回第3行计算的num-tasks和第4行的max-parallelism中的较小值，否则就返回num-tasks。TOPOLOGY-MAX-TASK-PARALLELISM表明当前Topology允许并行运行的Task的最大数目，这个参数主要是local模式下控制线程的数量，在真正的多机集群环境中一般不会使用。

下面我们举个例子来说明该计算过程。如果创建Topology时我们的设置如下：

```
1  TopologyBuilder builder = new TopologyBuilder();
2      builder.setSpout("spout1", new Spout1(), 1);
4      builder.setBolt("bolt", new Bolt1(), 5)
5          .shuffleGrouping("spout1");
6  Config conf = new Config();
7  conf.setDebug(false);
8  StormSubmitter.submitTopology("Test", conf, builder.createTopology());
```

对于组件bolt，我们设置它的并行度为5，没有为它设置TOPOLOGY-TASKS以及TOPOLOGY-MAX-TASK-PARALLELISM，所以对于该组件，调用component-parallelism方法的返回值为5。

如果修改创建过程，具体如下：

```
1 TopologyBuilder builder = new TopologyBuilder();
2     builder.setSpout("spout1", new Spout1(), 1);
4     builder.setBolt("bolt", new Bolt1(), 5)
5         .shuffleGrouping("spout1").setNumTasks(10);
6 Config conf = new Config();
7 conf.setDebug(false);
8 StormSubmitter.submitTopology("Test", conf, builder.createTopology());
```

对于组件bolt，我们除了设置它的并行度为5外，还设置TOPOLOGY-TASKS为10，没有为它设置TOPOLOGY-MAX-TASK-PARALLELISM，所以对于该组件，调用component-parallelism方法的返回值为10。

再次修改创建过程，具体如下：

```
1 TopologyBuilder builder = new TopologyBuilder();
2     builder.setSpout("spout1", new Spout1(), 1);
4     builder.setBolt("bolt", new Bolt1(), 5)
```

```
5        .shuffleGrouping("spout1").setNumTasks(10);
6 Config conf = new Config();
7 conf.setDebug(false);
8 conf.setMaxTaskParallelism(8);
9 StormSubmitter.submitTopology("Test", conf, builder.createTopology());
```

对于组件bolt，我们除了设置它的并行度为5外，还设置TOPOLOGY-TASKS为10，设置TOPOLOGY-MAX-TASK-PARALLELISM为8，所以对于该组件，调用component-parallelism方法的返回值为8。

6.8.3 compute-new-topology->executor->node+port

该方法根据系统当前已经存在的分配情况（上一轮分配后的结果），结合系统当前的运行情况找出需要进行任务分配的Topology集合，并为它们分配任务，计算出这一轮分配完之后的每个Topology对应的任务分配情况，也即<topology-id, <executor, [node,port]>>集合，其代码如下：

```
1 (defn compute-new-topology->executor->node+port [nimbus existing-assignments topologies scratch-
    topology-id]
2   (let [conf (:conf nimbus)
3       storm-cluster-state (:storm-cluster-state nimbus)
4       topology->executors (compute-topology->executors nimbus (keys existing-assignments))
5       ;; update the executors heartbeats first.
6       _ (update-all-heartbeats! nimbus existing-assignments topology->executors)
7       topology->alive-executors (compute-topology->alive-executors nimbus
8                                      existing-assignments
9                                      topologies
10                                     topology->executors
11                                     scratch-topology-id)
12      supervisor->dead-ports (compute-supervisor->dead-ports nimbus
13                                  existing-assignments
14                                  topology->executors
15                                  topology->alive-executors)
16      topology->scheduler-assignment (compute-topology->scheduler-assignment nimbus
17                                          existing-assignments
18                                          topology->alive-executors)
19
20      missing-assignment-topologies (->> topologies
21                                          .getTopologies
22                                          (map (memfn getId))
23                                          (filter (fn [t]
24                                              (let [alle (get topology->executors t)
25                                                    alivee (get topology->alive-executors t)]
26                                                  (or (empty? alle)
27                                                      (not= alle alivee)
28                                                      (< (-> topology->scheduler-assignment
29                                                             (get t)
30                                                             num-used-workers )
31                                                         (-> topologies (.getById t)
                                                               .getNumWorkers)
32                                                          ))
```

```
33                                                   ))))
34      all-scheduling-slots (->> (all-scheduling-slots nimbus topologies missing-assignment-
            topologies)
35                                (map (fn [[node-id port]] {node-id #{port}}))
36                                (apply merge-with set/union))
37
38      supervisors (read-all-supervisor-details nimbus all-scheduling-slots supervisor
            ->dead-ports)
39      cluster (Cluster. (:inimbus nimbus) supervisors topology->scheduler-assignment)
40      _(.schedule (:scheduler nimbus) topologies cluster)
41      new-scheduler-assignments (.getAssignments cluster)
42      ;; add more information to convert SchedulerAssignment to Assignment
43      new-topology->executor->node+port (compute-topology->executor->node+port
            new-scheduler-assignments)
44      supervisor-details (basic-supervisor-details-map storm-cluster-state)
45      reassign-details (StringBuilder.)]
46    (doseq [[topology-id executor->node+port] new-topology->executor->node+port
47             :let [old-executor->node+port (-> topology-id
48                                               existing-assignments
49                                               :executor->node+port)
50                     reassignment (filter (fn [[executor node+port]]
51                                           (and (contains? old-executor->node+port executor)
52                                                (not (= node+port (old-executor->node+port
                                                         executor)))))
53                                   executor->node+port)
54                     old-node->host (-> topology-id existing-assignments :node->host)
55                     new-node->host (->> executor->node+port vals (map first) set
56                                      (mapcat (fn [node]
57                                                (if-let [host (.getHostName (:inimbus
                                                    nimbus) supervisor-details node)][[node
                                                    host]])))
58                                     (into {}))]]
59      (when-not (empty? reassignment)
60          (let [new-slots-cnt (count (set (vals executor->node+port)))
61                  reassign-executors (keys reassignment)
62                  topology (.getById topologies  topology-id)]
63           (doseq [executor reassign-executors
64                     :let [executor-detail (ExecutorDetails. (first executor) (last executor))
65                             old-node+port (get old-executor->node+port executor)
66                             new-node+port (get executor->node+port executor)]]
67               (.append reassign-details
68                   (str "Executor " (get (.getExecutorToComponent topology) executor-detail) "
                       " executor-detail
69                       " in topology " (.getName topology)
70                       " is reassigned from " (get old-node->host (first old-node+port))
                            ":" (second old-node+port)
71                       " to " (get new-node->host (first new-node+port)) ":"
                            (second new-node+port) ". ")))
72          (log-message "Reassigning " topology-id " to " new-slots-cnt " slots")
73          (log-message "Reassign executors: " (vec reassign-executors)))))
74      new-topology->executor->node+port))
```

下面简要介绍该方法的几个参数。

- nimbus：nimbus-data对象。
- existing-assignment：当前已经进行的任务分配，是一个<topology-id, Assignment>集合。
- topologies：Topologies对象。
- scratch-topology-id：当前需要进行重新分配操作的topology-id。

下面简要介绍上述代码中各行代码的作用。

- 第2~3行从nimbus-data中获取:conf和:storm-cluster-state变量。
- 第4行调用compute-topolog->executors方法获取<topology-id,executors>集合（实际上可以通过对每个topology-id调用compute->executors方法获取到，compute->executors方法后面我们会介绍）。
- 第6行调用update-all-heartbeats！方法更新所有Topology的Executor的心跳信息。
- 第7~11行调用compute-topology->alive-executors获取<topology-id,alive-executors>集合。
- 第12~15行调用compute-supervisor->dead-ports获取<supervisor-id, dead-ports>集合。
- 第16~18行调用compute-topology->scheduler-assignment获取<topology-id,SchedulerAssignmentImpl>集合。
- 第20~33行计算missing-assignment-topologies。首先获取所有Topology的id，然后按照此条件进行过滤：该Topology的所有Executor为空或者该Topology的所有Executor不等于该Topology的活跃的Executor或者该Topology的num-used-workers小于其指定的num-workers。
- 第34~36行计算all-scheduling-slots，调用all-scheduling-slots方法得到集合，然后按照键进行分组，返回结果是<node-id(也即supervisor-id), ports>集合。
- 第38行调用read-all-supervisor-details方法获取<supervisor-id, SupervisorDetails>集合。
- 第39行利用nimbus-data中的:inimbus成员变量、第38行返回的<supervisor-id, SupervisorDetails>集合以及第16~18行返回的<topology-id, SchedulerAssignmentImpl>集合创建Cluster对象。
- 第40行调用nimbus-data中scheduler对象的schedule方法来对当前所有Topology进行任务调度。
- 第41行从cluster对象中获取重新调度完之后的所有Assignments作为new-scheduler-assignments，它是一个<topology-id, SchedulerAssignment>集合。
- 第43行调用compute-topology->executor->node+port方法将第41行返回的<topology-id, SchedulerAssignment>集合转化为<topology-id, {executor [node port]}>集合。
- 第44行调用basic-supervisor-details-map将ZooKeeper中记录的所有SupervisorInfo都转化为SupervisorDetails，返回<supervisor-id, SupervisorDetails>集合。
- 第45行创建一个Stringbuilder对象，后面会用它来记录重新分配信息。
- 在第46~73行中对于每一个Topology，通过将第43行返回的结果中的executor->node+port与ZooKeeper中保存的existing-assignment中的executor->node+port信息进行对比，计算出重新分配结果reassignment（条件是existing-assignment的executor->node+port信息中包含该

Executor，但是node+port与新的executor->node+port中的不同）。接下来，分别计算出old-node->host以及new-node->host，如果reassignment不为空，那么就会遍历这些被重新分配的Executor并打印出该Executor是由哪个old-host->port转化到了new-host->port。

- 第74行返回第43行计算出的结果，也即新的<topology-id, {executor [node port]}>集合。

6.8.4 compute-executors

该方法主要根据当前Topology设置的组件的并行度创建对应的Executor，其代码如下：

```
1 (defn- compute-executors [nimbus storm-id]
2    (let [conf (:conf nimbus)
3         storm-base (.storm-base (:storm-cluster-state nimbus) storm-id nil)
4         component->executors (:component->executors storm-base)
5         storm-conf (read-storm-conf conf storm-id)
6         topology (read-storm-topology conf storm-id)
7         task->component (storm-task-info topology storm-conf)]
8    (->> (storm-task-info topology storm-conf)
9         reverse-map
10        (map-val sort)
11        (join-maps component->executors)
12        (map-val (partial apply partition-fixed))
13        (mapcat second)
14        (map to-executor-id)
15        )))
```

该方法中涉及的参数如下所示。

- nimbus：nimbus-data对象。
- storm-id：topology-id。

下面简要介绍各行代码的作用。

- 第3行获取当前Topology的StormBase信息。
- 第4行从获取的StormBase信息中获取component->executors信息，它保存的是每个组件到它的并行度的映射。
- 第5行获取该Topology对应的Storm配置信息。
- 第6行获取该Topology对应的信息。
- 第7行调用storm-task-info方法获取<task-id, component-id>集合。该方法主要用于获取每个组件的TOPOLOGY-TASKS信息，并将其转化为<task-id, component-id>集合，其中task-id对该Topology的所有组件来讲是全局递增的。这里我们需要注意，前面在提交Topology时介绍过，normalize-topology方法会计算每个组件的并行度（依赖于TOPOLOGY-TASKS的设置或是用户设置的并行度），并会将计算出来的并行度更新到Storm配置项TOPOLOGY-TASKS中，所以这里获取到的TOPOLOGY-TASKS信息就是该组件的真正运行并行度。
- 第8~15行调用storm-task-info方法获取到<task-id, component-id>集合之后，首先将集合转换为<component-id, tasks>集合，然后对每个组件的任务集合按照升序排序，接下

来用第4行计算出来的<component-id, parallelism>信息跟排序后得到的<component-id, tasks>集合做join操作得到<component-id, [parallelism, tasks]>集合，然后调用partition-fixed方法对每一项的值[parallelism, tasks]进行处理，它主要是将这些tasks均匀分配到数目为parallelism的分区上，返回parallelism组[task-id1 task-id2 task-id3…]信息，最后取出这些信息并将其转换为一组Executor集合[[start-taskId, end-taskId] [start-taskId, end-taskId]...]，其中每个Executor集合都由一组连续的Task组成，所以Executor的表示方式为开始Task的TaskId和结束Task的TaskId。如果用户没有为每一个组件单独设置Task数目，那么组件的Task数目跟parallelism数目是相等的，这也意味着每个Executor实际上只有一个Task，也即start-taskId=end-taskId。

第 7 章

Scheduler

Scheduler是Storm的调度器，它负责为Topology分配当前集群中可用的资源。Storm定义了IScheduler接口，用户可以通过实现该接口来定义自己的Scheduler。Storm提供了3种Scheduler——EvenScheduler、DefaultScheduler和IsolationScheduler，下面简要介绍一下它们。

- EvenScheduler：会将系统中的可用资源均匀地分配给当前需要任务分配的多个Topology。
- DefaultScheduler：跟EvenScheduler基本一致，唯一的区别在于它会在为Topology分配任务之前先释放掉其他Topology不再需要的资源，然后调用EventScheduler方法为Topology均匀分配资源。
- IsolationScheduler：它提供了一种机制，使得用户可以单独为某些Topology指定它们需要的机器资源（机器数目）。用户需要在Storm配置项中指定这些信息（topology-name及其所需的机器数目），IsolationScheduler会优先对这些Topology分配任务，保证分配给某个Topology的机器只能运行这个特定的Topology，相当于这些Topology的运行环境是相互独立的。待这些指定的Topology分配完成之后，再调用DefaultScheduler，利用系统中剩余的资源为剩余的Topology进行任务分配。

7.1 IScheduler 接口

该接口是Storm定义的为集群当前所有Topology分配任务的接口，用户可以基于该接口实现自定义的Scheduler，它的定义如下：

```
1 public interface IScheduler {
2
3     void prepare(Map conf);
4
5    /**
6    * Set assignments for the topologies which needs scheduling. The new assignments is available
7    * through <code>cluster.getAssignments()</code>
8    *
9    *@param topologies all the topologies in the cluster, some of them need schedule. Topologies
          object here
10   *only contain static information about topologies. Information like assignments,
          slots are all in
11    *the <code>cluster</code>object.
```

```
12   *@param cluster the cluster these topologies are running in. <code>cluster</code> contains
        everything user
13   *need to develop a new scheduling logic. e.g. supervisors information, available slots,
        current
14   *assignments for all the topologies etc. User can set the new assignment for topologies using
15   *<code>cluster.setAssignmentById</code>
16   */
17   void schedule(Topologies topologies, Cluster cluster);
18 }
```

这个定义中，主要涉及两个方法，具体如下所示。

- prepare方法：它接收当前Nimbus的Storm配置作为参数，以进行一些初始化工作。
- scheduler方法：它是真正进行任务分配的方法。在Nimbus进行任务分配的时候会调用该方法。它的参数包括topologies和cluster：前者含有了当前集群中所有的Topology信息，后者则代表当前集群，其中包含用户自定义调度逻辑时所需的所有资源，包括Supervisor信息、当前可用的所有slot，以及任务分配情况等。

7.2　EvenScheduler

EvenScheduler是一个对资源进行均匀分配的调度器，它实现了IScheduler接口，相关代码如下：

```
1 (defn -prepare [this conf]
2   )
3
4 (defn -schedule [this ^Topologies topologies ^Cluster cluster]
5   (schedule-topologies-evenly topologies cluster))
```

可以看到，EvenScheduler是通过调用schedule-topologies-evenly方法来完成任务分配的，下面我们就来介绍这个方法以及它所调用的几个方法。

7.2.1　schedule-topolpgies-evenly

该方法会均匀地为Topology分配系统当前的资源，相关代码如下：

```
1 (defn schedule-topologies-evenly [^Topologies topologies ^Cluster cluster]
2     (let [needs-scheduling-topologies (.needsSchedulingTopologies cluster topologies)]
3          (doseq [^TopologyDetails topology needs-scheduling-topologies
4               :let [topology-id (.getId topology)
5                     new-assignment (schedule-topology topology cluster)
6                     node+port->executors (reverse-map new-assignment)]]
7         (doseq [[node+port executors] node+port->executors
8             :let [^WorkerSlot slot (WorkerSlot. (first node+port) (last node+port))
9                  executors (for [[start-task end-task] executors]
10                     (ExecutorDetails. start-task end-task))]]
11            (.assign cluster slot topology-id executors)))))
```

它的输入参数topologies和cluster前面已介绍过，这里不再赘述，下面简要介绍各行代码的作用。

- 第2行通过调用cluster对象的needsSchedulingTopologies方法来获取所有需要进行任务调度的Topology集合。这里，判断Topology是否需要进行任务调度的依据有两个：Topology设置的NumWorkers数目是否大于已经分配给该Topology的Worker数目，以及该Topology尚未分配的Executor的数目是否大于0。
- 第3~6行对需要进行任务调度的Topology中的每一个，首先获取其topology-id，然后调用schedule-topology方法（后面会介绍）获取计算得到的new-assignment（<executor, node+port>集合），最后将new-assignment的键和值颠倒获取<node+port, executors>集合。
- 第7~11行对于前面获取的<node+port, executors>集合中的每一项进行以下操作。
- 第8行用node和port信息构造WorkerSlot对象并将其作为slot。
- 第9~10行对于Executor集合中的每一项，构造ExecutorDetail对象，并返回一个ExecutorDetails集合作为executors。
- 第11行调用cluster的assign方法将计算出来的slot分配给与该Topology相对应的executors（第9~10行的计算结果）。

7.2.2 schedule-topology

该方法会根据集群当前的可用资源对Topology进行任务分配，其代码如下：


```
1 (defn- schedule-topology [^TopologyDetails topology ^Cluster cluster]
2   (let [topology-id (.getId topology)
3        available-slots (->> (.getAvailableSlots cluster)
4           (map #(vector (.getNodeId %) (.getPort %))))
5        all-executors (->> topology
6            .getExecutors
7            (map #(vector (.getStartTask %) (.getEndTask %)))
8            set)
9        alive-assigned (get-alive-assigned-node+port->executors cluster topology-id)
10       total-slots-to-use (min (.getNumWorkers topology)
11          (+ (count available-slots) (count alive-assigned)))
12       reassign-slots (take (- total-slots-to-use (count alive-assigned))
13          (sort-slots available-slots))
14       reassign-executors (sort (set/difference all-executors (set (apply concat (vals
            alive-assigned)))))
15       reassignment (into {}
16          (map vector
17              reassign-executors
18              ;; for some reason it goes into infinite loop without limiting the repeat-seq
19              (repeat-seq (count reassign-executors) reassign-slots)))]
20       (when-not (empty? reassignment)
21           (log-message "Available slots: " (pr-str available-slots))
22       )
23      reassignment))
```

- 第2行获取topology-id。
- 第3~4行调用cluster的getAvailableSlots方法获取集群当前可用的slot资源，并将其转换为<node, port>集合赋给available-slots变量。getAvailableSlots方法主要负责计算出当

前集群中还没有使用的Supervisor端口。

- 第5~8行将Topology中的ExecutorDetails集合转换为<start-task-id, end-task-id>集合，然后赋给all-executors并返回。
- 第9行调用get-alive-assigned-node+port->executors方法计算当前该Topology已经分得的资源情况，最后返回一个<node+port, executors>集合并将其保存到alive-assigned变量中。
- 第10~11行计算当前Topology可以使用的slot数目，将其作为total-slots-to-use变量的内容。该值取以下两者中的最小值。
  - Topology设置的Worker数目；
  - 当前的available-slots数目加上alive-assigned数目。
- 第12~13行对available-slots进行排序（这个排序方法会在后面介绍），计算需要分配的slot数目（total-slots-to-use减去alive-assigned的数目），最后从排序后的available-slots集合中按顺序取出这些slot作为reassign-slots。
- 第14行通过比较all-executors跟已经分配的Executor集合间的差异，获取需要进行分配的Executor集合，并将其作为reassign-executors。
- 第15~19行将前面计算出来的reassign-executors和reassign-slots进行关联，转换为<executor, slot>映射集合，并保存在reassignment中，这时有两种情况。
  - reassign-executors数目少于reassign-slots数目。这意味着当前集群中的可用资源比较多。比如reassign-executors是{e1,e2,e3}，而reassign-slots是{slot1,slot2,slot3,slot4,slot5}，那么就只取reassign-slots的前三项跟reassign-executors配对，返回{[e1,slot1],[e2, slot2], [e3, slot3]}。
  - reassign-executors数目大于reassign-slots数目，这意味着当前集群中的可用资源非常有限。比如reassign-executors是{e1,e2,e3,e4,e5,e6}，而reassign-slots是{slot1, slot2}，此时就会有多个Executor被分配到同一个slot上，返回的结果则可能是{[e1,slot1],[e2,slot2],[e3,slot1],[e4,slot2],[e5,slot1],[e6,slot2]}。
- 第20~22行判断reassignment是否为空，若不为空则打印日志记录当前可用的slot信息。
- 第23行返回计算得到的reassignment，它是一个<executor, [node, port]>集合。

7.2.3 get-alive-assigned-node+port->executors

这个方法用于获取Topology当前已经分得的资源，其代码如下：

```
1 (defn get-alive-assigned-node+port->executors [cluster topology-id]
2     (let [existing-assignment (.getAssignmentById cluster topology-id)
3          executor->slot (if existing-assignment
4              (.getExecutorToSlot existing-assignment)
5              {})
6          executor->node+port (into {} (for [[^ExecutorDetails executor ^WorkerSlot slot]
             executor->slot
```

```
7            :let [executor [(.getStartTask executor) (.getEndTask executor)]
8            node+port [(.getNodeId slot) (.getPort slot)]]]
9            {executor node+port}))
10       alive-assigned (reverse-map executor->node+port)]
11    alive-assigned))
```

- 第2行获取该Topology当前的assignment。
- 第3~5行判断当前的assignment是否为空，若不为空，则获取其中的<executor, slot>信息。
- 第6~9行将前面取得的<executor, slot>信息转换为<executor, [node,port]>集合。
- 第10行将前面的<executor, [node, port]>信息转换为<[node, port], executors>信息。
- 第11行返回算出的<[node, port], executors>信息。

7.2.4 sort-slots

该方法会对slot进行排序，其代码如下：

```
1 (defn sort-slots [all-slots]
2    (let [split-up (vals (group-by first all-slots))]
3         (apply interleave-all split-up)
4    ))
5
6 (defn interleave-all [& colls]
7    (if (empty? colls)
8        []
9        (let [colls (filter (complement empty?) colls)
10            my-elems (map first colls)
11            rest-elems (apply interleave-all (map rest colls))]
12    (concat my-elems rest-elems)
13    )))
```

该方法的参数是需要进行排序的slot列表，下面简要介绍各行代码的作用。

- 第2行将所有的slot按照键（也即supervisor-id）进行分组，然后返回分组之后的值的集合。
- 第3行调用interleave-all方法对前面返回的集合进行处理。
- 第6行定义了interleave-all方法。
- 第7行判断传入的colls集合是否为空，如果为空，直接返回空集合。
- 第9行过滤掉传入集合中的空元素。
- 第10行调用map first处理传入的colls集合。map first会遍历colls集合，获取其中每个集合的第一个元素，并将返回的结果保存在my-elems中。
- 第11行递归调用interleave-all处理剩下的集合。
- 第12 行将第10行和第11行的处理结果合并成一个集合返回。

1. sort-slots示例

下面我们来举例介绍这个sort-slots方法的执行过程。

假设我们传入的slot集合为([“a” 1] [“b” 1] [“c” 1] [“a” 2] [“b” 2] [“c” 2] [“a” 3] [“b” 3] [“c” 3])，

经过第2行处理后返回的结果是([[“a” 1] [“a” 2] [“a” 3]] [[“b” 1] [“b” 2] [“b” 3]] [[“c” 1] [“c” 2] [“c” 3]])。然后调用interleave-all继续对其进行处理，第10行返回的my-elems将是([“a” 1] [“b” 1] [“c” 1])。第11行递归调用传入的参数是([[“a” 2] [“a” 3]] [[“b 2”] [“b” 3]] [[“c” 2] [“c” 3]])，返回的rest-elems是([“a” 2] [“b” 2] [“c” 2] [“a” 3] [“b” 3] [“c” 3])。所以最终返回的结果是([“a” 1] [“b” 1] [“c” 1] [“a” 2] [“b” 2] [“c” 2] [“a” 3] [“b” 3] [“c” 3])。

2. sort-slots方法的问题

这个排序方法看上去好像是均匀分布的，但在实际应用中却可能导致集群中资源分配不均匀，即出现有的机器上所有的slot都被占用但是有的机器上的slot却一个也没被占用的现象。

从上面对sort-slots方法的介绍中也能很好地理解这一点。还是上面的这个例子，若Topology1提交的时候，当前集群中的资源都是空闲的，排序后返回的就是([“a” 1] [“b” 1] [“c” 1] [“a” 2] [“b” 2] [“c” 2] [“a” 3] [“b” 3] [“c” 3])。假设Topology1需要使用2个slot，分配完之后[“a” 1]和[“b” 1]就已经被占用了，此时我们提交Topology2，再次排序返回的结果将是([“a” 2] [“b” 2] [“c” 1] [“a” 3] [“b” 3] [“c” 2] [“c” 3])，这时如果Topology2需要4个slot，它就会使用([“a” 2] [“b” 2] [“c” 1] [“a” 3])。这样，系统中的分配情况便是机器a上的三个slot都被用完了，机器b上用掉了两个slot，而机器c上仅使用了一个。我们期望的情形却是均匀分配（暂不考虑CPU和内存），也即a、b、c机器上都有两个slot被占用。如何做到这一点呢？其实只要稍微改一下排序的算法，换为按照slot的port进行排序和分组，然后对每个分组内部分别进行排序（这一步可有可无）即可。

7.3　DefaultScheduler

DefaultScheduler是Storm默认的任务调度器，如果用户没有指定自己实现的调度器，Storm就会使用该调度器进行任务分配。其实现如下:

```
1 (defn -prepare [this conf]
2     )
3
4 (defn -schedule [this ^Topologies topologies ^Cluster cluster]
5     (default-schedule topologies cluster))
```

它的schedule方法会调用default-schedule方法来进行任务分配。下面我们详细介绍一下这个方法以及它所依赖的几个主要方法。

7.3.1　default-schedule

这个方法主要是计算当前集群中所有可供分配的slot资源，并判断当前已经分配给该Toplolgy的slot资源是否需要重新分配，然后通过这些信息明确哪些slot可被用来对新提交的Topology进行任务分配，最后调用EvenScheduler的schedule-topologies-evenly方法完成分配。其代码如下：

```
1 (defn default-schedule [^Topologies topologies ^Cluster cluster]
2     (let [needs-scheduling-topologies (.needsSchedulingTopologies cluster topologies)]
3         (doseq [^TopologyDetails topology needs-scheduling-topologies
```

```
4               :let [topology-id (.getId topology)
5                     available-slots (->> (.getAvailableSlots cluster)
6                       (map #(vector (.getNodeId %) (.getPort %))))
7                     all-executors (->> topology
8                        .getExecutors
9                        (map #(vector (.getStartTask %) (.getEndTask %)))
10                       set)
11                    alive-assigned (EvenScheduler/get-alive-assigned-node+port
                         ->executors cluster topology-id)
12                     can-reassign-slots (slots-can-reassign cluster (keys alive-assigned))
13                     total-slots-to-use (min (.getNumWorkers topology)
14                        (+ (count can-reassign-slots) (count available-slots)))
15                     bad-slots (if (> total-slots-to-use (count alive-assigned))
16                         (bad-slots alive-assigned (count all-executors) total-slots-to-use)
17                         [])]]
18              (.freeSlots cluster bad-slots)
19              (EvenScheduler/schedule-topologies-evenly (Topologies. {topology-id topology})
                   cluster))))
```

- 第2行调用cluster的needsSchedulingTopologies方法获取所有需要进行任务调度的Topology集合，详见前面有关EvenScheduler的介绍。
- 第3~19行是对每一个需要进行任务调度的Topology进行处理。
- 第4行获取topology-id。
- 第5~6行调用cluster的getAvailableSlots方法获取当前集群中所有可用的slot资源，并将其转换为<node, port>集合赋给available-slots变量。
- 第7~10行获取Topology的所有的Executor信息，并将其转换为<start-task-id,end-task-id>集合存入all-executors。
- 第11行调用EvenScheduler的get-alive-assigned-node+port->executors方法，计算当前Topology已经分得的任务信息，将返回的<[node,port], executor>信息保存到alive-assigned变量中。
- 第12行调用slots-can-reassign方法对alive-assigned的slot信息进行判断，选出其中可被重新分配的slot集合并保存到can-reassign-slots变量中。后面我们会介绍slots-can-reassign方法的具体实现。
- 第13~14行计算当前Topology所能使用的全部slot的数目，它取以下两个量中较小的值作为total-slots-to-use。
    - Topology的NumWorkers参数值。
    - available-slots的数目加上can-reassign-slots的数目。
- 第15~17行用于判断如果total-slots-to-use的数目大于当前已经分配的slot数目，则调用bad-slots方法计算所有可被释放的slot。后面我们会介绍bad-slots这个方法。
- 第18行调用cluster的freeSlots方法释放前面计算出来的bad-slots。
- 第19行调用EvenScheduler的schedule-topologies-evenly方法将系统中的资源均匀分配该Topology。

7

7.3.2 slots-can-reassign

这个方法从已经分配给当前Topology的资源中过滤出可以继续使用的资源，其代码如下：

```
1 (defn slots-can-reassign [^Cluster cluster slots]
2     (->> slots
3         (filter
4             (fn [[node port]]
5                 (if-not (.isBlackListed cluster node)
6                     (if-let [supervisor (.getSupervisorById cluster node)]
7                         (.contains (.getAllPorts supervisor) (int port))
8                         ))))))
```

该方法的输入参数有两个。

- cluster：cluster对象，前面我们介绍过。
- slots：已经分配的slot资源，是一个<node, port>集合。

该方法将对传入的slots集合进行过滤，选出其中仍然可以继续使用的slot组成新的集合。过滤方法是先判断slot的node信息是否存在于集群的黑名单里。如果不在，继续判断slot的port信息是否在与node相对应的Supervisor的所有可用端口列表中，如果在，那么这个slot就可以继续使用。

7.3.3 bad-slots

这个方法用于计算一个Topology已经被分配的资源中哪些是不再需要的，其代码如下：

```
1 (defn- bad-slots [existing-slots num-executors num-workers]
2    (if (= 0 num-workers)
3        '()
4        (let [distribution (atom (integer-divided num-executors num-workers))
5                keepers (atom {})]
6           (doseq [[node+port executor-list] existing-slots :let [executor-count
                 (count executor-list)]]
7              (when (pos? (get @distribution executor-count 0))
8                  (swap! keepers assoc node+port executor-list)
9                  (swap! distribution update-in [executor-count] dec)
10                 ))
11          (->> @keepers
12               keys
13               (apply dissoc existing-slots)
14               keys
15               (map (fn [[node port]]
16                        (WorkerSlot. node port)))))))
```

首先介绍该方法的参数。

- existing-slots：已经分配给Topology的资源，它是一个<[node, port], executors>集合。
- num-executors：Topology的所有Executor（包括已经分配的和未分配的）。
- num-workers：Topology可使用的全部slot数目。

下面简要介绍各行代码的作用。

- 第2行判断num-workers是否为0。如果是，表明当前没有可供该Topology使用的slot，这时返回一个空集合。
- 第4~5行定义distribution集合和keepers集合。distribution集合通过调用integer-divided方法生成，它实际上就是将num-executors均匀地分配到num-workers中。例如，若num-executors是10，num-workers是4，这时计算出来的结果将是一个<executor-count, worker-count>集合（这个例子中是{<2,2>,<3,2>}）。executor-count代表某个worker被分配了多少个Executor，worker-count则代表有多少个这样的Worker。所以对于这个例子，计算结果表明有2个Worker上会被分别分配2个Executor，2个Worker上会被分别分配3个Executor。keepers集合默认为空集合。
- 第6~10行对于传入的existing-slots中的每一项，计算其对象的executor-count，然后从前面计算的distribution集合中以该executor-count作为键去获取值。如果获取的值大于0，意味着存在这样的分配，也即存在至少一个Worker上有executor-count个Executor，那么，当前这个分配信息便会继续维持。这时，会将该<[node, port], executors>信息放入keepers中，同时将distribution中该executor-count的对应值减一。
- 第11~16行从existing-slots中移除keepers中记录的需要继续维持的分配情况。如果移除完之后还存在slot信息，表明这些slot可以被释放掉，于是将其转换为WorkerSlot对象集合并返回。

7

7.4 IsolationScheduler

这是Storm本身提供的另外一种调度器，它的主要目的是提供一种机制来确保集群中的某些Topology有足够的运行资源。要使用这个调度器，需要在配置项中给定isolation.scheduler.machines参数，其内容是一个从Topology的名字到指定机器数目的映射，另外还需要将storm.scheduler设置为backtype.storm.scheduler.IsolationScheduler。下面我们来介绍它的实现，其代码如下：

```
1 (defn -prepare [this conf]
2   (container-set! (.state this) conf))
3
4 (defn -schedule [this ^Topologies topologies ^Cluster cluster]
5   (let [conf (container-get (.state this))
6            orig-blacklist (HashSet. (.getBlacklistedHosts cluster))
7            iso-topologies (isolated-topologies conf (.getTopologies topologies))
8            iso-ids-set (->> iso-topologies (map #(.getId ^TopologyDetails %)) set)
9            topology-worker-specs (topology-worker-specs iso-topologies)
10            topology-machine-distribution (topology-machine-distribution conf iso-topologies)
11            host-assignments (host-assignments cluster)]
12      (doseq [[host assignments] host-assignments]
13         (let [top-id (-> assignments first second)
14               distribution (get topology-machine-distribution top-id)
15               ^Set worker-specs (get topology-worker-specs top-id)
16               num-workers (count assignments)
17            ]
```

```
18            (if (and (contains? iso-ids-set top-id)
19                   (every? #(= (second %) top-id) assignments)
20                   (contains? distribution num-workers)
21                   (every? #(contains? worker-specs (nth % 2)) assignments))
22                (do (decrement-distribution! distribution num-workers)
23                    (doseq [[_ _ executors] assignments] (.remove worker-specs executors))
24                    (.blacklistHost cluster host))
25                (doseq [[slot top-id _] assignments]
26                   (when (contains? iso-ids-set top-id)
27                       (.freeSlot cluster slot)
28                       ))
29                )))
30
31    (let [host->used-slots (host->used-slots cluster)
32           ^LinkedList sorted-assignable-hosts (host-assignable-slots cluster)]
33      ;; TODO: can improve things further by ordering topologies in terms of who needs the
            least workers
34      (doseq [[top-id worker-specs] topology-worker-specs
35                :let [amts (distribution->sorted-amts (get topology-machine-distribution
                      top-id))]]
36          (doseq [amt amts
37                   :let [[host host-slots] (.peek sorted-assignable-hosts)]]
38          (when (and host-slots (>= (count host-slots) amt))
39              (.poll sorted-assignable-hosts)
40              (.freeSlots cluster (get host->used-slots host))
41              (doseq [slot (take amt host-slots)
42                        :let [executors-set (remove-elem-from-set! worker-specs)]]
43                  (.assign cluster slot top-id executors-set))
44          (.blacklistHost cluster host))
45          )))
46
47     (let [failed-iso-topologies (->> topology-worker-specs
48                                    (mapcat (fn [[top-id worker-specs]]
49                                                (if-not (empty? worker-specs) [top-id])
50                                                )))]
51         (if (empty? failed-iso-topologies)
52             ;; run default scheduler on non-isolated topologies
53             (-<> topology-worker-specs
54                 allocated-topologies
55                 (leftover-topologies topologies <>)
56                 (DefaultScheduler/default-schedule <> cluster))
57             (do
58                (log-warn "Unable to isolate topologies " (pr-str failed-iso-topologies) ".
                     No machine had enough worker slots to run the remaining workers for these
                     topologies. Clearing all other resources and will wait for enough resources
                     for isolated topologies before allocating any other resources.")
59               ;; clear workers off all hosts that are not blacklisted
60              (doseq [[host slots] (host->used-slots cluster)]
61                 (if-not (.isBlacklistedHost cluster host)
62                    (.freeSlots cluster slots)
63                    )))
64            ))
65     (.setBlacklistedHosts cluster orig-blacklist)
66     ))
```

- 第2行调用container-set!方法将Storm配置项信息保存到state中。
- 第5行获取state中保存的Storm配置项。
- 第6行获取调度前cluster对象保存的主机名的黑名单集合。由于在调度的执行过程中会对该集合进行修改，因此这里需提前保存原始值，待调度完成后再将它恢复原值。
- 第7行调用isolated-topologies方法获取传入的Topologies对象中需要单独分配资源的Topology信息。至于哪些Topology需要进行单独分配，则由前面提到的isolation.scheduler.machines参数来指定。isolated-topologies返回的是一个Topology Details集合，该集合会被保存到iso-topologies变量中。
- 第8行获取iso-topologies中每一项的topology-id，并将返回的topology-id集合保存到iso-ids-set变量中。
- 第9行调用topology-worker-specs方法对iso-topologies进行处理。它返回一个<topology-id, List<Set>>集合，其中List<Set>中的每一项都是一个Worker上的Executor集合。Executor的结构是[start-task-id, end-task-id]。
- 第10行调用topology-machine-distribution方法获取与每一个需要单独进行任务分配的Topology对应的机器分布信息，它也是由配置项中的isolation.scheduler.machines参数所指定的。该方法的返回结果是一个<topology-id, HashMap<worker-count, machine-count>>集合，其中哈希表记录了该Topology的所有Worker在指定机器上的数量分布情况。键代表有几个Worker，值代表有几台机器会被分配到该键所指定的数量的Worker。
- 第11行调用host-assignments方法获取当前集群中机器资源的分配情况。它返回的是按照主机名进行分组之后的信息，每个分组里面的信息是一个<slot, topology-id, executors>集合。返回的结果将被保存在host-assignments中。
- 第12~29行对前面获取到的host-assignments中的每一项依次进行处理。
  - 第13行获取assignments中第一项的topology-id信息。
  - 第14行从第10行的计算结果中获取与topology-id相对应的机器分布信息，即一个哈希表。
  - 第15行从第9行的计算结果中获取与topology-id相对应的Executor信息。
  - 第16行获取当前主机上被分配的任务数目并将其作为num-workers。
  - 第18~29行首先判断当前主机上的所有assignment是否满足以下条件。
    - 第8行计算出来的iso-ids-set中是否包含第13行获取的topology-id。
    - 当前host上的所有assignment是否都属于由第13行获取的topology-id所指定的Topology。
    - 第14行计算出来的哈希表中是否包含以第16行计算出来的num-workers为键的项。
    - 第15行计算出来的Executor集合是否包含了当前主机所分得任务的所有Executor信息。

    如果这4个条件全部满足，那么这个主机符合分配条件，不需要再对它进行分配，只需进行以下操作。
  - 调用decrement-distribution!方法更新第14行的结果中num-workers对应的机器数目，将它的值减一，即已经有一台机器符合这个分配条件。

  - 更新第15行结果的Executor集合，将当前这台机器上所有满足条件的Executor从该集合中移除，只对剩下的Executor进行分配。
  - 将满足条件的机器名加入到cluster对象的黑名单中，以确保不会再对这台机器进行任何的任务分配。

  如果前面4条中有任何一个条件不满足，那么就遍历该机器上所有的任务分配，判断它们对应的topology-id是否属于iso-ids-set集合。如果是，则调用cluster的freeSlot方法将与该任务相对应的slot释放掉。

- 第31行调用host->used-slots函数获取当前集群中的<host, used-slots>集合，将结果保存在host->used-slots变量中。
- 第32行调用host-assignable-slots方法获取当前集群中可用的slot资源集合，并将结果保存在sorted-assignable-hosts变量中。这个信息是一个<host, slots>集合，且它是按照slot的数目降序排列的。
- 第34~45行对第9行结果<topology-id, List<executors>>中的每一项进行处理。首先调用distribution->sorted-amts方法计算与Topology对应的每台机器上待分配的Worker数目集合（按照降序排列），并将该集合保存在amts变量中。接下来，对amts中的每一项进行处理。首先获取sorted-assignable-hosts中的第一个[host, host-slots]元素（此时并不会将元素移除，这里取第一个元素是因为sorted-assignable-hosts是按照机器可用的slot数目降序排列的，而amts中保存的Topology所需的Worker数目集合也是降序排列的，这就确保了只要检查sorted-assignable-hosts中的第一个元素就可以得知amt是否能满足所需的Worker数目）。如果host-slots不为空，并且host-slots的数目大于等于该Topology需要的数目，表明该机器可被用于分配。这时才会真正将sorted-assignable-hosts的第一个元素移除，并调用cluster的freeSlots方法将该主机上所有正在使用的slot释放掉。接下来，从该机器的host-slots列表中取出前amt个slot，其中每一个slot都对应一个Worker，每一个slot都会被分配给与该Topology对应的一组属于同一个Worker的Executor集合（第9行计算出来的）。最后，将该机器的主机信息加入到cluster的黑名单列表中，防止该机器再次被用于进行任务分配。
- 第47~50行用于判断所有需要单独进行任务分配的Topology是否都已经分配完，并返回没有分配完的topology-id列表作为failed-iso-topologies。
- 第51~64行判断failed-iso-topologies集合是否为空。如果为空，代表单独分配都已经完成，接下来要做的是从当前所有需要进行任务分配的Topology中移除这些需要单独分配并且已经分配完成的Topology，然后调用DefaultScheduler的default-schedule方法对剩余的Topology进行任务分配。由于在进行单独任务分配的过程中已经将成功分配的机器加入到了cluster的黑名单列表，所以这里对剩余的Topology进行分配时只会使用不在黑名单列表中的机器。如果failed-iso-topologies集合不为空，表明这些需要单独进行任务分配的Topology没有全部分配完成。这时会打印日志，记录下哪些Topology（也即failed-iso-topologies集合）还没有被分配，然后遍历当前所有已经被分配任务的机器，

如果该机器不在cluster的黑名单列表中，就释放掉它目前占用的资源。这一步的目的是为了确保下次为这些没有完成任务分配的Topology进行分配时有足够的资源。

- 第65行将cluster的黑名单恢复到进行任务调度之前的值，也即前面第6行保存的初始值。

7.5 调度示例

前面介绍了3种调度策略，下面我们举一个实际的例子来看一下不同调度器是如何进行任务分配的。

假设当前集群中有6台机器，每台机器上的可用端口均为6700、6701、6702和6703，且当前集群中没有正在运行的Topology，初始状态如图7-1所示。

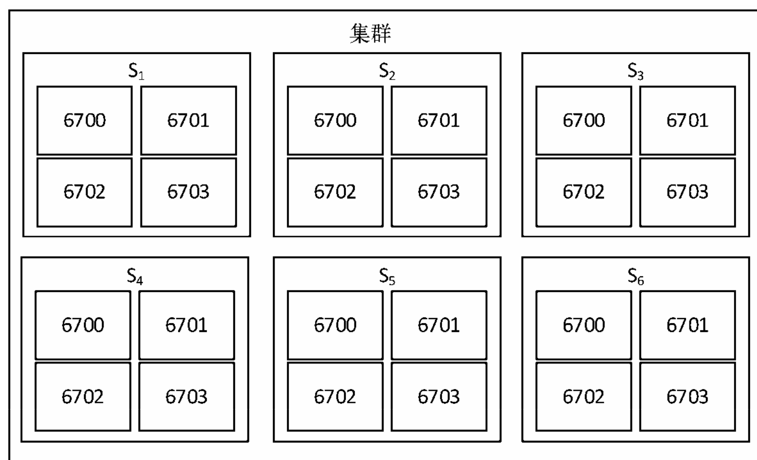


图7-1 集群初始状态

表7-1是接下来我们会依次提交的3个Topology。

表7-1 Scheduler测试Topology

Topology	Worker数目	Executor数目	Task数目
T-1	3	8	16
T-2	5	10	10
T-3	8	16	16

鉴于正常情况下EvenScheduler和DefaultScheduler的调度结果是一致的，我们把这两种方式放在一起介绍，下面就看一下不同的调度器对分配结果的影响。

7.5.1 EvenScheduler和DefaultScheduler

在这一节中，我们会详细介绍分别提交T-1、T-2以及T-3之后的任务调度以及资源分配情况。

7

1. 提交T-1

提交T-1后的调度过程如下。

- 可用的slot列表经过`sort-slots`方法处理后的结果是：{[S_1 6700][S_2 6700][S_3 6700][S_4 6700][S_5 6700][S_6 6700][S_1 6701][S_2 6701][S_3 6701][S_4 6701][S_5 6701][S_6 6701][S_1 6702][S_2 6702][S_3 6702][S_4 6702][S_5 6702][S_6 6702][S_1 6703][S_2 6703][S_3 6703][S_4 6703][S_5 6703][S_6 6703]}。
- `compute-executors`方法计算完后得到的Executor列表为：{[1 2][3 4][5 6][7 8][9 10][11 12][13 14][15 16]}。
- 8个Executor在3个Worker上的分布情况是[3，3，2]。

分配结果如下：

- {[1 2][3 4][5 6]}→[S_1 6700]
- {[7 8][9 10][11 12]}→[S_2 6700]
- {[13 14][15 16]}→[S_3 6700]

此时集群中的任务分配情况如图7-2所示。

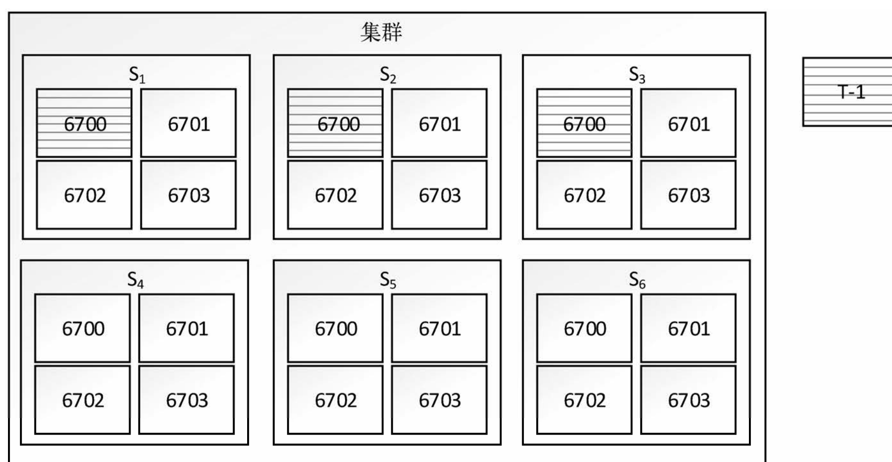


图7-2　提交完T-1后集群中的任务分配情况

2. 提交T-2

提交T-2后的调度过程如下。

- 可用的slot列表经过`sort-slots`方法处理后的结果是：{[S_1 6701][S_2 6701] [S_3 6701][S_4 6700][S_5 6700][S_6 6700] [S_1 6702][S_2 6702][S_3 6702][S_4 6701][S_5 6701][S_6 6701][S_1 6703][S_2 6703][S_3 6703][S_4 6702][S_5 6702][S_6 6702][S_4 6703][S_5 6703][S_6 6703]}。
- `compute-executors`方法计算完后得到的Executor列表为：{[1 1][2 2][3 3][4 4][5 5][6 6][7 7][8 8][9 9][10 10]}。
- 10个Executor在5个Worker上的分布情况是[2, 2, 2, 2, 2]。

分配结果如下：

- ❑ {[1 1][2 2]}→[S_1 6701]
- ❑ {[3 3][4 4]}→[S_2 6701]
- ❑ {[5 5][6 6]}→[S_3 6701]
- ❑ {[7 7][8 8]}→[S_4 6700]
- ❑ {[9 9][10 10]}→[S_5 6700]

此时集群中的任务分配情况如图7-3所示。

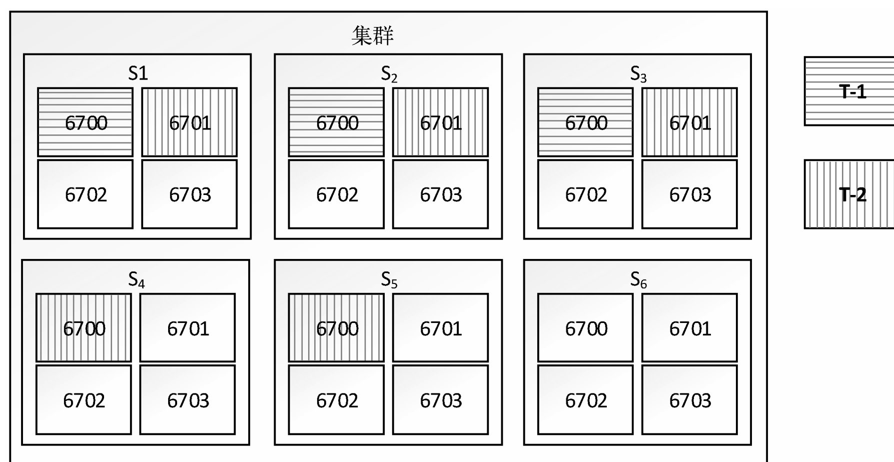


图7-3　提交完T-2后集群中的任务分配情况

3. 提交T-3

提交T-3后的调度过程如下。

- ❑ 可用的slot列表经过sort-slots方法处理后的结果是：{[S_1 6702][S_2 6702][S_3 6702][S_4 6701][S_5 6701][S_6 6700] [S_1 6703][S_2 6703][S_3 6703][S_4 6702][S_5 6702][S_6 6701][S_4 6703][S_5 6703][S_6 6702][S_6 6703]}。
- ❑ compute-executors方法计算完后得到的Executor列表为：{[1 1][2 2][3 3][4 4][5 5][6 6][7 7][8 8][9 9][10 10][11 11][12 12][13 13][14 14][15 15][16 16]}。
- ❑ 16个Executor在8个Worker上的分布情况是[2, 2, 2, 2, 2, 2, 2, 2]。

分配结果如下：

- ❑ {[1 1][2 2]}→[S_1 6702]
- ❑ {[3 3][4 4]}→[S_2 6702]
- ❑ {[5 5][6 6]}→[S_3 6702]
- ❑ {[7 7][8 8]}→[S_4 6701]
- ❑ {[9 9][10 10]}→[S_5 6701]
- ❑ {[11 11][12 12]}→[S_6 6700]

- {[13 13][14 14]}→[S_1 6703]
- {[15 15][16 16]}→[S_2 6703]

此时集群中的资源分配情况如图7-4所示。

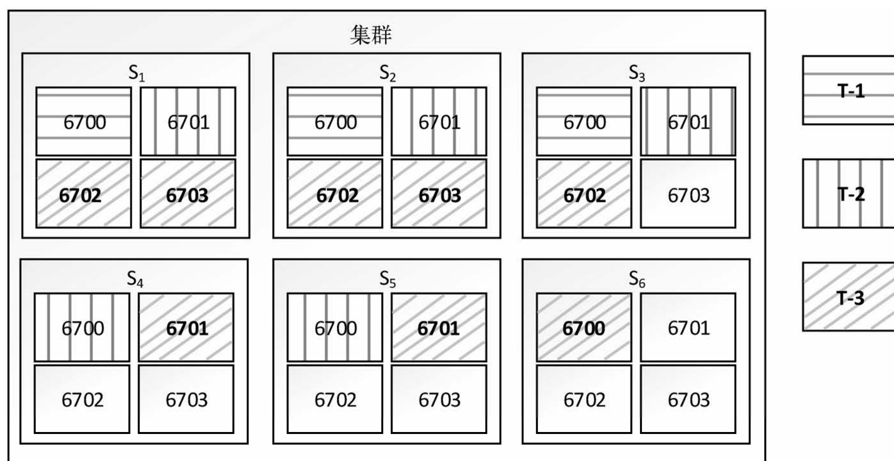


图7-4　提交完T-3后集群中的最终分配结果

由此也可以看出，目前的分配策略是有问题的，最终的任务分布并不均匀。

7.5.2　IsolationScheduler

假设我们将T-3设置为需要进行单独分配的Topology，并设置其所需要的机器数目为3，下面来看一下它的分配情况。

提交T-1和T-2时的任务分配情况跟前面使用DefaultScheduler和EvenScheduler进行分配是一样的，此时集群的状态如图7-3所示。

紧接着我们提交T-3，调度器发现这是一个需要单独进行分配的Topology，所以首先会计算它的任务分配情况，计算过程如下。

- compute-executors方法计算完后得到的Executor列表：{[1 1][2 2][3 3][4 4][5 5][6 6][7 7][8 8][9 9][10 10][11 11][12 12][13 13][14 14][15 15][16 16]}。
- 16个Executor在8个Worker上的分布情况是[2, 2, 2, 2, 2, 2, 2, 2]。
- 设定它需要3台机器（即3个Supervisor），总共有8个Worker，所以Worker在机器上的分布情况按降序排列是[3, 3, 2]。

接下来检查当前集群中是否有已经分配给该Topology的资源。如果有，进一步检查当前机器上是否只有属于该Topology的Worker，且Worker数目是否符合我们计算的分布情况（这里就是有3个或2个）。如果不符合要求，就将当前机器上所有分配给需要单独进行分配的Topology的资源都释放掉，本例中这一步没做任何事情。接下来获取系统中所有的可用资源，并按照主机名进行分组操作，然后将分组后的可用资源按数目从大到小排列。这一步返回的结果是：

{{S_6, {[S_6 6700][S_6 6701][S_6 6702][S_6 6703]}},

{S_5, {[S_5 6701][S_5 6702][S_5 6703]}},

{S_4, {[S_4 6701][S_4 6702][S_4 6703]}},

{S_3, {[S_3 6702][S_3 6703]}},

{S_2, {[S_2 6702][S_2 6703]}},

{S_1, {[S_1 6702][S_1 6703]}}}

也即S_6上有4个可用资源，S_5、S_4上有3个可用资源，S_3、S_2和S_1上都有两个可用资源。

我们已经计算得知Worker的分布是[3，3，2]。首先判断可用机器S_6上能否启动3个该Topology的Worker。此处可以，所以会将S_6算作符合分配条件的机器。同时若S_6上已经有了在运行的任务，则会被释放掉，然后在这台机器上为该Topology分配3个Worker：

- ❑ {[1 1][2 2]}→[S_6 6700]
- ❑ {[3 3][4 4]}→[S_6 6701]
- ❑ {[5 5][6 6]}→[S_6 6702]

同时会把S_6机器加入到cluster对象的黑名单中。

接下来判断S_5上能否启动3个属于该Topology的Worker，这里也是可以的。所以它也算符合条件的机器，因此释放掉它上面已经启动的属于T-2的一个Worker，然后在该机器上为该Topology分配3个Worker：

- ❑ {[7 7][8 8]}→[S_5 6700]
- ❑ {[9 9][10 10]}→[S_5 6701]
- ❑ {[11 11][12 12]}→[S_5 6702]

同时把S_5机器加入到cluster对象的黑名单中。

同理，接下来判断S_4上是否能启动两个该Topology的Worker，这里也是可以的。所以S_4同样也是符合分配条件的机器，因此释放掉它上面已经启动的属于T-2的一个Worker，然后在该机器上为该Topology分配两个Worker：

- ❑ {[13 13][14 14]}→[S_4 6700]
- ❑ {[15 15][16 16]}→[S_4 6701]

最后，将S_4也加入到cluster对象的黑名单中。

至此就完成了对T_3的任务分配。接下来通过调用DefaultScheduler为之前释放掉的T-2的两个Worker重新分配资源，最终的分配如图7-5所示。

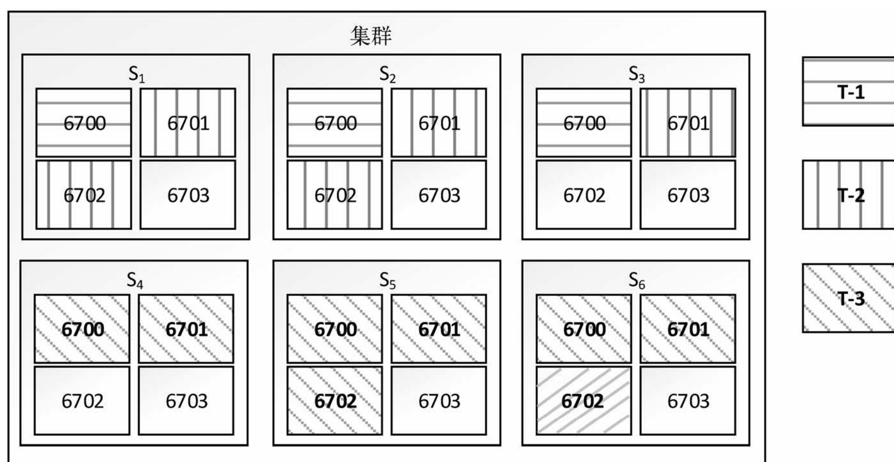


图7-5　IsolationScheduler最终分配情况

第8章 Scheduler

8

Supervisor可理解为单机任务调度器，它负责监听Nimbus的任务调度，启动相应的Worker对Nimbus分配的任务进行处理。同时，它也会监测由它启动的Worker的运行状态，一旦发现有Worker处于非正常状态，就会杀掉该Worker，并将原来分配给该Worker的任务交还Nimbus进行重新分配。

8.1 与 Supervisor 相关的数据结构

首先我们来介绍一些Supervisor中定义的常用方法与数据结构，具体如下所示。

- standalone-supervisor方法：它会创建一个实现了ISupervisor接口的对象，其中定义了一些常用方法。
- supervisor-data方法则创建了一个映射集合，这是Supervisor中一个非常重要的数据结构，它包含了很多会被其他方法共享的成员。
- Supervisor使用LocalState在本地保存了自己的ID信息、LocalAssignment信息以及有效的从Worker到端口号的映射关系。

8.1.1 standalone-supervisor

这个方法返回一个实现了ISupervisor接口的对象，它可以用来获取和创建Supervisor的id，也可以用来获取配置给当前Supervisor的所有可用端口列表。下面我们仔细看一下它的实现，其代码如下：

```
1 (defn standalone-supervisor []
2     (let [conf-atom (atom nil)
3              id-atom (atom nil)]
4          (reify ISupervisor
5               (prepare [this conf local-dir]
6                   (reset! conf-atom conf)
7                   (let [state (LocalState. local-dir)
8                            curr-id (if-let [id (.get state LS-ID)]
9                                          id
10                                         (generate-supervisor-id))]
11                       (.put state LS-ID curr-id)
12                       (reset! id-atom curr-id))
13                 )
```

```
14          (confirmAssigned [this port]
15              true)
16          (getMetadata [this]
17              (doall (map int (get @conf-atom SUPERVISOR-SLOTS-PORTS))))
18          (getSupervisorId [this]
19              @id-atom)
20          (getAssignmentId [this]
21              @id-atom)
22          (killedWorker [this port]
23              )
24          (assigned [this ports]
25              ))))
```

- 第5~13行定义的prepare方法用来保存当前Supervisor使用的配置信息，获取和创建Supervisor的id，以及将该标识保存在id-atom中。
- 第14~15行定义的confirmAssigned方法用于确认一个端口是否已被分配，目前直接返回true。
- 第16~17行定义的getMetadata方法从配置中获取该Supervisor的所有可用端口号信息。
- 第18~19行定义的getSupervisorId和第20～21行定义的getAssignmentId方法的实现目前是相同的，返回的都是该Supervisor的id信息。
- 第22～25行定义的killedWorker和assigned方法目前没有使用。

8.1.2　Supervisor的数据

supervisor-data方法定义了整个Supervisor代码的共享数据结构，其中包括了很多常用的成员变量。它在Supervisor启动时创建，并在接下来的过程中在许多方法中作为参数传入，相关代码如下：

```
(defn supervisor-data [conf shared-context ^ISupervisor isupervisor]
    {:conf conf
    :shared-context shared-context
    :isupervisor isupervisor
    :active (atom true)
    :uptime (uptime-computer)
    :worker-thread-pids-atom (atom {})
    :storm-cluster-state (cluster/mk-storm-cluster-state conf)
    :local-state (supervisor-state conf)
    :supervisor-id (.getSupervisorId isupervisor)
    :assignment-id (.getAssignmentId isupervisor)
    :my-hostname (if (contains? conf STORM-LOCAL-HOSTNAME)
         (conf STORM-LOCAL-HOSTNAME)
         (local-hostname))
    :curr-assignment (atom nil) ;; used for reporting used ports when heartbeating
    :timer (mk-timer :kill-fn (fn [t]
        (log-error t "Error when processing event")
        (halt-process! 20 "Error when processing an event")
        ))
    })
```

我们分别介绍一下这些成员变量的作用。

- :conf是Supervisor启动时使用的配置信息。
- :shared-context可用于传递一些与上下文相关的信息，目前没有使用（默认为nil）。
- :isupervisor是一个实现了ISupervisor接口的对象，这里就是通过前面介绍的standalone- supervisor方法获取的一个实例。
- :active表明Supervisor的运行状态。
- :uptime表明Supervisor迄今为止的运行时间。
- :worker-thread-pids-atom存储Supervisor启动的所有Worker的进程号集合。
- :storm-cluster-state用来获取或存储与Storm集群相关的元数据信息。
- :local-state创建该Supervisor的LocalState存储对象。
- :supervisor-id为当前Supervisor的身份标识。
- :assignment-id目前等同于:supervisor-id。
- :my-hostname记录了当前Supervisor的主机名信息。
- :curr-assignment记录了当前Supervisor已经使用的端口号信息。
- :timer是Supervisor创建的计时器，用于执行周期性的任务。

8.1.3 本地存储数据

为了确保Supervisor在失败重启后能继续正常运行，它在本地存储了一些重要的数据结构，这些数据都是通过LocalState保存的，下面简要介绍一下。

8

1. Supervisor Id

Supervisor将自己的id保存在路径STORM-LOCAL-DIR/supervisor/isupervisor/下，保存时使用的键是supervisor-id。这样就算Supervisor服务被重启，Supervisor仍然可以通过LocalState获取自己的supervisor-id。

2. LocalAssignment

它记录了Supervisor上从storm-id（topology-id）到分配给该Supervisor的所有Executor的对应关系，相关代码如下：

```
(defrecord LocalAssignment [storm-id executors])
```

Supervisor将这些信息保存在路径STORM-LOCAL-DIR/supervisor/localstate/下，保存时使用的键是local-assignments。Supervisor每次跟Nimbus同步时，都会将自己本地保存的目前分配与从Nimbus获取的最新分配相比较，若发现分配给自己的任务发生变化，则做出相应处理并将本地记录更新为目前分配。Supervisor服务重启后，仍然可以通过LocalState获取自己上次被分配的任务，从而可以继续正常工作。

3. Approved Workers

它主要保存当前Supervisor机器上有效的<worker-id, port>映射集合。Supervisor将这些信息也保存在路径STORM-LOCAL-DIR/supervisor/localstate/下，保存时使用的键是approved-workers。

Supervisor每次跟它启动的Worker进程同步时，首先会通过获取本地保存的任务分配信息得到最新的任务分配情况，接下来会检查当前已经启动的Worker是否都还活着（心跳一直更新），然后根据这些信息计算出哪些Worker需要关闭、哪些Worker需要启动，从而计算出新的approved-worker集合，并将更新后的内容保存在LocalState数据中。Supervisor服务重启后，仍可获得由它启动的正在运行的Worker列表，从而继续正常工作。

8.2　Supervisor 中的线程

Supervisor中有3个线程，一个计时器线程以及两个事件线程。计时器线程主要负责维持心跳，也即更新ZooKeeper中保存的当前Supervisor的最新状态，同时也负责每隔一段时间将事件线程要执行的事件添加到其对应的队列中；在两个事件线程中，一个负责与Nimbus同步任务，另一个则负责根据任务变化同步管理Worker进程。

8.2.1　计时器线程

计时器线程会每隔固定的时间间隔运行一遍下面定义的函数，其中时间间隔由参数SUPERVISOR-HEARTBEAT-FREQUENCY-SECS定义（默认值是5秒）：

```
1 fn [] (.supervisor-heartbeat!
2    (:storm-cluster-state supervisor)
3    (:supervisor-id supervisor)
4    (SupervisorInfo. (current-time-secs)
5        (:my-hostname supervisor)
6        (:assignment-id supervisor)
7        (keys @(:curr-assignment supervisor))
8         ;; used ports
9        (.getMetadata isupervisor)
10       (conf SUPERVISOR-SCHEDULER-META)
11       ((:uptime supervisor))))
```

该方法会更新保存在ZooKeeper中的该Supervisor的信息，这样保证了Nimbus在进行新一轮任务分配时，会及时得知当前集群中各个Supervisor的最新状态。

除此之外，计时器线程还会每隔10秒将mk-synchronize-supervisor方法加入到同步Nimbus任务的事件线程中，每隔SUPERVISOR-MONITOR-FREQUENCY-SECS（默认是3秒）将sync-processes方法加入到同步管理Worker进程的事件线程中。

8.2.2　同步Nimbus任务的线程

该线程通过不断执行mk-synchronize-supervisor函数来保证Supervisor与Nimbus的任务同步，以及时获取Nimbus分配给该Supervisor的新任务，并移除那些已经分配但不再需要的旧任务。在这个处理过程，Supervisor会将每次同步后的LocalAssignment信息更新到LocalState中。

下面我们看一下这个函数的实现，其代码如下：

```
1 (defn mk-synchronize-supervisor [supervisor sync-processes event-manager processes-event-manager]
2   (fn this []
3       (let [conf (:conf supervisor)
4                 storm-cluster-state (:storm-cluster-state supervisor)
5                 ^ISupervisor isupervisor (:isupervisor supervisor)
6                 ^LocalState local-state (:local-state supervisor)
7                 sync-callback (fn [& ignored] (.add event-manager this))
8                 storm-code-map (read-storm-code-locations storm-cluster-state sync-callback)
9                 downloaded-storm-ids (set (read-downloaded-storm-ids conf))
10                all-assignment (read-assignments
11                    storm-cluster-state
12                    (:assignment-id supervisor)
13                    sync-callback)
14                new-assignment (->> all-assignment
15                    (filter-key #(.confirmAssigned isupervisor %)))
16                assigned-storm-ids (assigned-storm-ids-from-port-assignments new-assignment)
17                existing-assignment (.get local-state LS-LOCAL-ASSIGNMENTS)]
18      (log-debug "Synchronizing supervisor")
19      (log-debug "Storm code map: " storm-code-map)
20      (log-debug "Downloaded storm ids: " downloaded-storm-ids)
21      (log-debug "All assignment: " all-assignment)
22      (log-debug "New assignment: " new-assignment)
23
24      (doseq [[storm-id master-code-dir] storm-code-map]
25          (when (and (not (downloaded-storm-ids storm-id))
26                  (assigned-storm-ids storm-id))
27              (log-message "Downloading code for storm id "
28                  storm-id
29                  " from "
30                  master-code-dir)
31              (download-storm-code conf storm-id master-code-dir)
32              (log-message "Finished downloading code for storm id "
33                  storm-id
34                  " from "
35                  master-code-dir)
36              ))
37
38      (log-debug "Writing new assignment "
39          (pr-str new-assignment))
40      (doseq [p (set/difference (set (keys existing-assignment))
41                (set (keys new-assignment)))]
42          (.killedWorker isupervisor (int p)))
43      (.assigned isupervisor (keys new-assignment))
44      (.put local-state
45              LS-LOCAL-ASSIGNMENTS
46             new-assignment)
47          (reset! (:curr-assignment supervisor) new-assignment)
48          (if on-windows? (shutdown-disallowed-workers supervisor))
49          (doseq [storm-id downloaded-storm-ids]
50              (when-not (assigned-storm-ids storm-id)
51                 (log-message "Removing code for storm id "
52                    storm-id)
53                 (try
54                 (rmr (supervisor-stormdist-root conf storm-id))
55                 (catch Exception e (log-message (.getMessage e))))
```

```
56                  ))
57          (.add processes-event-manager sync-processes)
58          )))
```

首先介绍该方法涉及的参数。

- supervisor：supervisor-data对象。
- sync-processes：mk-supervisor中定义的用于同步当前Supervisor启动的Worker的方法。它负责关闭被移除的Worker，以及启动新分配的Worker。
- event-manager：mk-supervisor中定义的运行mk-synchronize-supervisor方法的线程。
- processes-event-manager：mk-supervisor中定义的运行sync-processes方法的线程。

下面简述各行代码的作用。

- 第3~6行依次获取supervisor-data中的配置信息、storm-local-cluster、isupervisor和local-state对象，并将它们分别保存到对应的临时变量中。
- 第7行定义sync-callback方法，它会简单地将mk-synchronize-supervisor方法再次加入到同步Nimbus任务的事件线程中执行。这个方法主要用来在进行ZooKeeper操作时设置回调方法，另外当ZooKeeper中的内容发生变化时也会调用该方法。
- 第8行通过调用read-storm-code-locations方法获取<storm-id, master-code-location>集合信息作为storm-code-map。在read-storm-code-locations方法中，会先调用storm-cluster-state的assignments方法获取ZooKeeper中当前保存的storm-id信息，这里就会使用第7行定义的回调方法。当ZooKeeper中的任务分配信息发生变化时（增加或删除Topology），就会调用该回调方法及时处理这些变化。
- 第9行通过调用read-downloaded-storm-ids方法获取当前已下载Topology所对应的Storm id信息。
- 第10~13行调用read-assignments方法获取分配给当前Supervisor的任务信息，并返回一个<port, LocalAssignment>集合作为all-assignment的内容。
- 第14~15行对前面计算出来的all-assignment进行过滤，只保留其中已经被确认分配的<port, LocalAssignment>信息作为new-assignment（在当前的实现中，这一步什么都没做，所以实际上new-assignment跟all-assignment是一样的）。
- 第16行调用assigned-storm-ids-from-port-assignments方法获取new-assignment的storm-id信息作为assigned-storm-ids。
- 第17行获取local-state中保存的LS-LOCAL-ASSIGNMENTS信息作为existing-assignment，它是一个<port, Assignment>集合。
- 第24~36行调用download-storm-code下载当前已经分配给该Supervisor但还没有下载到本地的Topology信息。
- 第40~42行首先计算出属于existing-assignment但不属于new-assignment的端口信息，然后调用isupervisor的killedWorker方法关闭这些端口上对应的Worker（目前，killedWorker方法是一个空方法）。

- 第43行调用isupervisor的assigned方法设置新分配的端口信息。
- 第44~46行更新local-state中保存的LS-LOCAL-ASSIGNMENTS信息，将其设置为new-assignment。
- 第47行更新supervisor-data中的:curr-assignment信息，将其设置为new-assignment。
- 第48行判断当前操作系统是否为Windows，若是，则调用shutdown-disallowed-workers方法将状态为:disallowed的Worker关闭。
- 第49~56行移除本地已经下载的但已不在assigned-storm-ids中的Topology信息。
- 第57行将sync-processes方法添加到processes-event-manager中执行，这么做是为了确保当Supervisor与Nimbus同步完任务分配之后，立即调用sync-processes方法同步Worker进程。

8.2.3 管理Worker进程的线程

该线程通过不断执行sync-processes函数来管理由该Supervisor启动的所有Worker进程，这种管理包括关闭当前不处于:valid状态的Worker，以及启动新分配给该Supervisor的Worker。在这个处理过程中，Supervisor会将每次同步完后的Approved Workers信息更新到LocalState中。

下面我们看一下这个方法的具体实现，相关代码如下：

```
1 (defn sync-processes [supervisor]
2   (let [conf (:conf supervisor)
3           ^LocalState local-state (:local-state supervisor)
4           assigned-executors (defaulted (.get local-state LS-LOCAL-ASSIGNMENTS) {})
5           now (current-time-secs)
6           allocated (read-allocated-workers supervisor assigned-executors now)
7           keepers (filter-val
8                (fn [[state _]] (= state :valid))
9                allocated)
10          keep-ports (set (for [[id [_ hb]] keepers] (:port hb)))
11          reassign-executors (select-keys-pred (complement keep-ports) assigned-executors)
12          new-worker-ids (into
13              {}
14              (for [port (keys reassign-executors)]
15                [port (uuid)]))
16          ]
17
18    (log-debug "Syncing processes")
19    (log-debug "Assigned executors: " assigned-executors)
20    (log-debug "Allocated: " allocated)
21    (doseq [[id [state heartbeat]] allocated]
22        (when (not= :valid state)
23           (log-message
24           "Shutting down and clearing state for id " id
25           ". Current supervisor time: " now
26           ". State: " state
27           ", Heartbeat: " (pr-str heartbeat))
28           (shutdown-worker supervisor id)
29           ))
30    (doseq [id (vals new-worker-ids)]
31       (local-mkdirs (worker-pids-root conf id)))
32    (.put local-state LS-APPROVED-WORKERS
```

```
33        (merge
34              (select-keys (.get local-state LS-APPROVED-WORKERS)
35              (keys keepers))
36              (zipmap (vals new-worker-ids) (keys new-worker-ids))
37              ))
38    (wait-for-workers-launch
39        conf
40        (dofor [[port assignment] reassign-executors]
41            (let [id (new-worker-ids port)]
42                 (log-message "Launching worker with assignment "
43                  (pr-str assignment)
44                  " for this supervisor "
45                  (:supervisor-id supervisor)
46                  " on port "
47                  port
48                  " with id "
49                  id
50                  )
51               (launch-worker supervisor
52                   (:storm-id assignment)
53                   port
54                   id)
55               id)))
56    ))
```

在这个方法中，参数supervisor是一个supervisor-data对象，下面简述各行代码的作用。

- ❑ 第4行从local-state中获取LS-LOCAL-ASSIGNMENTS集合（一个<port, Assignment>集合），默认返回一个空集合。
- ❑ 第5行获取当前系统时间。
- ❑ 第6行调用read-allocated-workers获取当前已经分配的Worker信息，返回的是一个<worker-id, <worker state, worker heartbeat>>集合，其中记录了与当前分配的Worker相对应的状态和心跳信息。
- ❑ 第7~9行对第6行返回的<worker-id, <worker-state, worker heartbeat>>集合进行过滤，只保留其中worker-state为:valid的Worker。
- ❑ 第10行从第7~9行中过滤出来的Worker心跳信息中获取其所对应的端口信息，并将其作为要保留的端口信息。
- ❑ 第11行利用前面计算出来的要保留的端口信息，从第4行获取到的当前已经被分配的Executor中确定需要被重新分配的Executor信息，返回值是一个<port, Assignment>集合。
- ❑ 第12~15行利用第11行计算出来的需重新分配的Executor信息，为其中的每个端口创建一个新的worker-id，并返回<port, worker-id>集合。
- ❑ 第21~29行对第6行返回的集合中的每一项进行处理，判断其worker-state是否为:valid，如果不是就打印日志并调用shutdown-worker方法关闭该Worker。
- ❑ 第30~31行对于第12~15行返回的<port, worker-id>集合中的每一个worker-id创建pid文件夹，其路径是STORM-LOCAL-DIR/workers/<worker-id>/pids/。

- 第32~37行更新local-state中存储的LS-APPROVED-WORKERS信息。更新的方式是：首先获取当前local-state中存储的LS-APPROVED-WORKERS信息，然后根据第7~9行计算出的所有仍然有效的Worker的worker-id进行过滤，只保留继续有效的Worker信息，接下来将第12~15行计算出来的新的<port, worker-id>集合转换为<worker-id, port>集合，最后将这两部分进行合并，并将合并后的结果作为新的LS-APPROVED-WORKERS信息保存到local-state中。
- 第38~56行首先对第11行中返回的重新分配的<port, Assignment>集合中的每一项，根据端口信息从第12~15行返回的<port, worker-id>中获取其对应的worker-id，然后打印启动Worker的日志，调用launch-worker方法启动Worker，并返回这些启动的Worker的worker-id，最后调用wait-for-workers-launch方法等待这些Worker启动起来。

8.3 启动 Supervisor

launch方法用于启动Supervisor服务，它主要通过调用mk-supervisor方法来启动Supervisor服务，相关代码如下：

```
1 (defn -launch [supervisor]
2   (let [conf (read-storm-config)]
3        (validate-distributed-mode! conf)
4        (mk-supervisor conf nil supervisor)))
```

在这个方法中，传入的参数是一个实现了ISupervisor接口的对象，这里默认使用standalone-supervisor函数返回的对象。下面简述各行代码的作用。

- 第2行读取当前Supervisor启动时所需的配置信息。
- 第3行调用validate-distributed-mode!方法，根据前面读取的配置信息验证系统当前是否处于分布式模式。
- 第4行调用mk-supervisor方法创建并启动Supervisor。

下面我们来看一下mk-supervisor方法的具体实现，相关代码如下：

```
1 (defserverfn mk-supervisor [conf shared-context ^ISupervisor isupervisor]
2     (log-message "Starting Supervisor with conf " conf)
3     (.prepare isupervisor conf (supervisor-isupervisor-dir conf))
4     (FileUtils/cleanDirectory (File. (supervisor-tmp-dir conf)))
5     (let [supervisor (supervisor-data conf shared-context isupervisor)
6         [event-manager processes-event-manager :as managers] [(event/event-manager false)
           (event/event-manager false)]
7      sync-processes (partial sync-processes supervisor)
8          synchronize-supervisor (mk-synchronize-supervisor supervisor sync-processes event
             -manager processes-event-manager)
9      heartbeat-fn (fn [] (.supervisor-heartbeat!
10                           (:storm-cluster-state supervisor)
11                           (:supervisor-id supervisor)
12                           (SupervisorInfo. (current-time-secs)
13                               (:my-hostname supervisor)
14                               (:assignment-id supervisor)
```

```
15                                    (keys @(:curr-assignment supervisor))
16                                    ;; used ports
17                                    (.getMetadata isupervisor)
18                                    (conf SUPERVISOR-SCHEDULER-META)
19                                    ((:uptime supervisor)))))]
20      (heartbeat-fn)
21      ;; should synchronize supervisor so it doesn't launch anything after being down (optimization)
22      (schedule-recurring (:timer supervisor)
23          0
24          (conf SUPERVISOR-HEARTBEAT-FREQUENCY-SECS)
25          heartbeat-fn)
26      (when (conf SUPERVISOR-ENABLE)
27          ;; This isn't strictly necessary, but it doesn't hurt and ensures that the machine
                stays up
28          ;; to date even if callbacks don't all work exactly right
29          (schedule-recurring (:timer supervisor) 0 10 (fn [] (.add event-manager synchronize
               -supervisor)))
30          (schedule-recurring (:timer supervisor)
31              0
32              (conf SUPERVISOR-MONITOR-FREQUENCY-SECS)
33              (fn [] (.add processes-event-manager sync-processes))))
34      (log-message "Starting supervisor with id " (:supervisor-id supervisor) " at host "
            (:my-hostname supervisor))
35      (reify
36            Shutdownable
37            (shutdown [this]
38                (log-message "Shutting down supervisor " (:supervisor-id supervisor))
39                (reset! (:active supervisor) false)
40                (cancel-timer (:timer supervisor))
41                (.shutdown event-manager)
42                (.shutdown processes-event-manager)
43                (.disconnect (:storm-cluster-state supervisor)))
44      SupervisorDaemon
45      (get-conf [this]
46          conf)
47      (get-id [this]
48          (:supervisor-id supervisor))
49      (shutdown-all-workers [this]
50          (let [ids (my-worker-ids conf)]
51              (doseq [id ids]
52                 (shutdown-worker supervisor id)
53                 )))
54      DaemonCommon
55      (waiting? [this]
56          (or (not @(:active supervisor))
57              (and
58                  (timer-waiting? (:timer supervisor))
59                  (every? (memfn waiting?) managers)))
60          ))))
```

首先介绍mk-supervisor中涉及的参数。

- conf：启动该Supervisor所用的Storm配置项。
- shared-context：启动该Supervisor所用的上下文信息，目前没有使用。

- isupervisor：实现了ISupervisor接口的对象，这里默认使用standalone-supervisor。

下面简述各行代码的作用。

- 第3行调用isupervisor的prepare方法。这里，传入的Supervisor id的保存路径为：STORM-LOCAL-DIR/supervisor/isupervisor。
- 第4行清理supervisor-tmp-dir，路径为：STORM-LOCAL-DIR/supervisor/tmp。
- 第5行创建supervisor-data，这个对象前面介绍过。
- 第6~33行创建并启动我们前面介绍过的3个线程。
- 第35~60行返回一个实现了Shutdownable、SupervisorDaemon和DaemonCommon接口的对象，目前这个没有用到。

8.4　关闭 Supervisor

关闭Supervisor时，需要释放掉当前Supervisor所占用的资源，相关代码如下：

```
1 (shutdown [this]
2     (log-message "Shutting down supervisor " (:supervisor-id supervisor))
3     (reset! (:active supervisor) false)
4     (cancel-timer (:timer supervisor))
5     (.shutdown event-manager)
6     (.shutdown processes-event-manager)
7     (.disconnect (:storm-cluster-state supervisor)))
```

- 第3行将当前Supervisor的运行状态设置为false。
- 第4行关闭计时器线程。
- 第5行关闭该Supervisor与Nimbus同步任务的线程。
- 第6行关闭该Supervisor管理Worker进程的线程。
- 第7行释放掉与ZooKeeper的连接。

8.5　重要方法介绍

这一节我们主要介绍Supervisor中几个重要的辅助方法，具体如下所示。

- launch-worker
- read-allocated-workers
- wait-for-worker-launch
- shutdown-worker
- dowload-storm-code

8.5.1　launch-worker

launch-worker方法用于启动Worker进程。Storm提供了两种调用模式：Local和分布式模式。

在LocalCluster模式运行时，就使用Local模式；而在实际的集群中运行时，则使用分布式模式。下面分别介绍一下这两种模式。

1. 分布式模式

在分布式模式下，主要通过构造JVM命令行参数来启动Worker进程，相关代码如下：

```
1 (defmethod launch-worker
2     :distributed [supervisor storm-id port worker-id]
3     (let [conf (:conf supervisor)
4           stormroot (supervisor-stormdist-root conf storm-id)
5           stormjar (supervisor-stormjar-path stormroot)
6           storm-conf (read-supervisor-storm-conf conf storm-id)
7           classpath (add-to-classpath (current-classpath) [stormjar])
8           childopts (.replaceAll (str (conf WORKER-CHILDOPTS) " " (storm-conf TOPOLOGY
                 -WORKER-CHILDOPTS))
9                      "%ID%"
10                     (str port))
11          jmx-port-offset (System/getProperty "storm.worker.jmxremote.port.offset")
12          jmx-port (if (nil? jmx-port-offset) (+ port 1000) (+ port (Integer. jmx-port-offset)))
13          logfilename (str "worker-" port ".log")
14          command (str "java -server " childopts
15                   " -Djava.library.path=" (conf JAVA-LIBRARY-PATH)
16                   " -Dlogfile.name=" logfilename
17                   " -Dstorm.home=" (System/getProperty "storm.home")
18                   " -Dstorm.log.dir=" (System/getProperty "storm.log.dir")
19                   " -Dlogback.configurationFile=" (System/getProperty "logback
                        .configurationFile")
20                   " -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.
                         authenticate=false"
21                   " -Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management
                         .jmxremote.port=" jmx-port
22                   " -cp " classpath " backtype.storm.daemon.worker "
23                   (java.net.URLEncoder/encode storm-id) " " (:assignment-id supervisor)
24                   " " port " " worker-id)]
25         (log-message "Launching worker with command: " command)
26         (launch-process command :environment {"LD_LIBRARY_PATH" (conf JAVA-LIBRARY-PATH)})
27         ))
```

- 第2行的`:distributed`表明这是分布式模式的实现，它的参数有如下4个。
  - `supervisor`：supervisor-data对象。
  - `storm-id`：与该Worker对应的`storm-id`（`topology-id`）。
  - `port`：该Worker使用的端口号。
  - `worker-id`：该Worker的`id`，全局唯一。
- 第4行调用`supervisor-stormdist-root`方法获取Supervisor机器上的`stormroot`路径，它的结构是：STORM-LOCAL-DIR/supervisor/stormdist/<storm-id>/。
- 第5行调用`supervisor-stormjar-path`方法获取Supervisor机器上保存jar包的路径，它的结构是：STORM-LOCAL-DIR/supervisor/stormdist/<storm-id>/stormjar.jar。
- 第6行调用`read-supervisor-storm-conf`方法获取启动Worker时所需的配置信息，这些配置

信息的路径是：STORM-LOCAL-DIR/supervisor/stormdist/<storm-id>/stormconf.ser。

- 第7行调用add-to-classpath方法将第5行获取的stormjar路径添加到当前的class-path路径下，并将其保存在classpath变量中当前的classpath路径是指调用System.getProperty("java.class.path")获取的路径。
- 第8~10行将WORKER-CHILDOPTS和TOPOLOGY-WORKER-CHILDOPTS拼接起来，并将其中的%ID%替换为真正使用的port，最后将替换后的内容保存在childopts变量中。
- 第11行获取设置好的JMX端口偏移量。
- 第12行获取使用的JMX端口。如果第11行获取到了JMX端口偏移量（不为空），那么JMX使用的端口设置为port+jmx偏移量，否则将其设置为port+1000。JMX端口通常用于获取内置的运行统计。
- 第13行获取Worker日志文件的名字，它的格式是：worker-<port>.log。
- 第14~24行构造启动Worker的命令，它需要利用前面构造出来的childopts、java.library.path、logfile、home、log.dir、日志的配置文件、JMX端口，classpath、storm-id、supervisor-id、port以及worker-id等信息。
- 第26行调用launch-process方法，用构造好的命令启动Worker进程。

2. Local模式

在Local模式下，我们采用线程模拟进程的方式来启动Worker，相关代码如下：

```
1 (defmethod launch-worker
2    :local [supervisor storm-id port worker-id]
3    (let [conf (:conf supervisor)
4              pid (uuid)
5              worker (worker/mk-worker conf
6                          (:shared-context supervisor)
7                          storm-id
8                          (:assignment-id supervisor)
9                          port
10                         worker-id)]
11    (psim/register-process pid worker)
12    (swap! (:worker-thread-pids-atom supervisor) assoc worker-id pid)
13    ))
```

- 第2行的:local表明这段代码是Local模式的实现，其参数跟分布式模式是一样的。
- 第4行构造一个pid。Local模式使用线程来模拟进程，所以它使用process_simulator.clj中的方法来保存当前启动的Worker信息。这个pid作为Worker的身份标识。
- 第5~10行调用worker.clj的mk-worker方法启动Worker线程，返回构造的Worker对象。
- 第11行将Worker对象注册到process_simulator.clj中保存。
- 第12行将<worker-id, pid>的对应信息更新到supervisor-data的:worker-thread-pids-atom变量中。

第11行和第12行操作的目的是保存当前启动Worker的对应信息，这些信息在后面关闭Worker时用到。

8.5.2　read-allocated-workers

该方法用于获取Worker及其对应的心跳信息，并根据心跳信息判断该Worker的状态，相关代码如下：

```
1 (defn read-allocated-workers
2    [supervisor assigned-executors now]
3    (let [conf (:conf supervisor)
4             ^LocalState local-state (:local-state supervisor)
5             id->heartbeat (read-worker-heartbeats conf)
6             approved-ids (set (keys (.get local-state LS-APPROVED-WORKERS)))]
7        (into
8             {}
9             (dofor [[id hb] id->heartbeat]
10               (let [state (cond
11                    (not hb)
12                    :not-started
13                    (or (not (contains? approved-ids id))
14                        (not (matches-an-assignment? hb assigned-executors)))
15                      :disallowed
16                    (> (- now (:time-secs hb))
17                          (conf SUPERVISOR-WORKER-TIMEOUT-SECS))
18                      :timed-out
19                    true
20                             :valid)]
21                    (log-debug "Worker " id " is " state ": " (pr-str hb) " at supervisor time
                          -secs " now)
2                    [id [state hb]]
23                    ))
24          )))
```

首先介绍该方法中的参数。

- supervisor：supervisor-data对象。
- assigned-executors：分配给该Supervisor的任务信息，是一个<port, LocalAssignment>集合。
- now：调用该方法时的当前时间。

下面简述各行代码的作用。

- 第3~4行获取supervisor-data中的配置信息以及local-state对象。
- 第5行调用read-worker-heartbeats方法获取<worker-id, heartbeat>信息。
- 第6行获取当前local-state中保存的LS-APPROVED-WORKERS信息，获取其中的worker-id集合作为approved-ids。
- 第7~23行对第5行返回的id->heartbeat中的每一项，根据其心跳信息判断当前状态。
  - 如果没有心跳信息，设置状态为未启动。
  - 如果该worker-id不在approved-ids列表中，或者assigned-executors中没有分配给该Worker的任务，或者心跳中保存的storm-id和Executor信息与新分配的任务（Local Assignment）中的storm-id和Executor信息不一致，则设置Worker状态为非法。

- 如果最新的心跳时间距离当前时间（now）的间隔大于SUPERVISOR-WORKER-TIMEOUT-SECS（默认值是30秒），则设置Worker状态为超时；
- 否则设置Worker状态为合法。

将状态信息记录在state变量中，然后返回一个<worker-id, <state, heartbeat>>集合。

8.5.3 wait-for-worker-launch

该方法在启动Worker时被调用，它会保证直到Worker成功启动起来后才返回，相关代码如下：

```
1 (defn- wait-for-worker-launch [conf id start-time]
2     (let [state (worker-state conf id)]
3          (loop []
4                (let [hb (.get state LS-WORKER-HEARTBEAT)]
5                          (when (and
6                              (not hb)
7                              (<
8                              (- (current-time-secs) start-time)
9                              (conf SUPERVISOR-WORKER-START-TIMEOUT-SECS)
10                             ))
11               (log-message id " still hasn't started")
12               (Time/sleep 500)
13               (recur)
14               )))
15        (when-not (.get state LS-WORKER-HEARTBEAT)
16            (log-message "Worker " id " failed to start")
17  )))
```

首先介绍该方法的3个参数。

- conf：启动Worker时需要的配置信息。
- id：Worker的id信息。
- start-time：启动该Worker的时间。

下面简述各行代码的作用。

- 第2行调用worker-state方法获取（或创建）该Worker的local-state对象，并将其保存到state变量中，它的路径为：STORM-LOCAL-DIR/workers/<worker-id>/heatbeats/。
- 第3~14行是一个循环操作，它用于获取state中保存的LS-WORKER-HEARTBEAT信息。如果没有获取到heartbeat信息，且当前启动时间还小于SUPERVISOR-WORKER-START-TIMEOUT-SECS（默认值是120秒），则认为Worker还没启动起来，此时会打印日志，然后休眠500毫秒并继续循环操作。
- 第15~16行判断Worker是否有心跳信息，如果没有就代表启动失败，此时会打印日志。由于在启动时间超过SUPERVISOR-WORKER-START-TIMEOUT-SECS（默认值是120秒）时也会退出前面的循环操作，这一步则确保了Worker能够真正地启动起来。

8.5.4　shutdown-worker

该方法用于关闭Worker进程并清理Worker的本地文件夹，相关代码如下：

```
1 (defn shutdown-worker [supervisor id]
2     (log-message "Shutting down " (:supervisor-id supervisor) ":" id)
3     (let [conf (:conf supervisor)
4               pids (read-dir-contents (worker-pids-root conf id))
5               thread-pid (@(:worker-thread-pids-atom supervisor) id)]
6        (when thread-pid
7               (psim/kill-process thread-pid))
8        (doseq [pid pids]
9               (ensure-process-killed! pid)
10              (try
11                 (rmpath (worker-pid-path conf id pid))
12                 (catch Exception e)) ;; on windows, the supervisor may still holds the lock on
                   the worker directory
13            )
14       (try-cleanup-worker conf id))
15    (log-message "Shut down " (:supervisor-id supervisor) ":" id))
```

首先介绍该方法的两个参数。

- supervisor：supervisor-data对象。
- id：worker-id。

下面简述各行代码的作用。

- 第4行调用read-dir-contents方法获取路径STORM-LOCAL-DIR/workers/<worker-id>/pids/下面的进程号信息。
- 第5行根据worker-id从supervisor-data的:worker-thread-pids-atom中获取进程号信息，这里只有在LocalCluster模式时才会有进程号信息。
- 如果第5行获取到了进程号信息，第6~7行就调用process_simulator.clj中的kill-process方法关闭启动的Worker线程。
- 第8~13行对当前Worker中的每一个进程号调用ensure-process-killed!方法确保该进程号对应的进程已经被杀掉，然后尝试删除该进程号的本地路径STORM-LOCAL-DIR/workers/<worker-id>/pids/<pid>/。
- 第14行尝试清理Worker的本地目录：首先删除STORM-LOCAL-DIR/workers/<worker-id>/heartbeats/目录，接着删除STORM-LOCAL-DIR/workers/<worker-id>/pids/目录，最后删除STORM-LOCAL-DIR/workers/<worker-id>/目录。

8.5.5　download-storm-code

这个方法用于从Nimbus下载与分配给当前Supervisor的任务相对应的Topology信息。跟launch-worker方法类似，该方法也有两种模式——Local模式和分布式模式，下面我们分别给予介绍。

1. 分布式模式

分布式模式使用Nimbus提供的下载方法来获取所需的Topology信息，相关代码如下：

```
1 (defmethod download-storm-code
2     :distributed [conf storm-id master-code-dir]
3     ;; Downloading to permanent location is atomic
4     (let [tmproot (str (supervisor-tmp-dir conf) file-path-separator (uuid))
5              stormroot (supervisor-stormdist-root conf storm-id)]
6         (FileUtils/forceMkdir (File. tmproot))
7
8         (Utils/downloadFromMaster conf (master-stormjar-path master-code-dir) (supervisor
             -stormjar-path tmproot))
9         (Utils/downloadFromMaster conf (master-stormcode-path master-code-dir)
             (supervisor-stormcode-path tmproot))
10        (Utils/downloadFromMaster conf (master-stormconf-path master-code-dir)
             (supervisor-stormconf-path tmproot))
11        (extract-dir-from-jar (supervisor-stormjar-path tmproot) RESOURCES-SUBDIR tmproot)
12        (FileUtils/moveDirectory (File. tmproot) (File. stormroot))
13    ))
```

- 第2行的`:distributed`表明这是分布式模式的实现，它的参数有如下3个。
  - `conf`：Supervisor使用的配置信息。
  - `storm-id`：要下载的Topology的`id`。
  - `master-code-dir`：Nimbus机器上保存该Topology信息的文件夹。
- 第4行构造一个临时的root文件夹来存放新下载的Topology信息，它的路径是STORM-LOCAL-DIR/supervisor/tmp/<uuid>/。
- 第5行创建真正存储Topology下载信息的文件夹，其路径是STORM-LOCAL-DIR/supervisor/stormdist/<storm-id>/。
- 第6行创建第4行的临时文件夹。
- 第8行将stormjar.jar文件下载到临时文件夹中。
- 第9行将stormcode.ser文件下载到临时文件夹中。
- 第10行将stormconf.ser文件下载到临时文件夹中。
- 第11行调用`extract-dir-from-jar`方法将stormjar.jar中的资源文件夹提取出来放到临时文件夹中。
- 第12行将临时文件夹里的内容全部移到真正的存储目录中。

2. Local模式

在Local模式下，我们将从本机复制所需的Topology信息，相关代码如下：

```
1 (defmethod download-storm-code
2     :local [conf storm-id master-code-dir]
3     (let [stormroot (supervisor-stormdist-root conf storm-id)]
4         (FileUtils/copyDirectory (File. master-code-dir) (File. stormroot))
5         (let [classloader (.getContextClassLoader (Thread/currentThread))
6              resources-jar (resources-jar)
```

```
7            url (.getResource classloader RESOURCES-SUBDIR)
8            target-dir (str stormroot file-path-separator RESOURCES-SUBDIR)]
9            (cond
10                resources-jar
11                (do
12                    (log-message "Extracting resources from jar at " resources-jar " to "
                          target-dir)
13                    (extract-dir-from-jar resources-jar RESOURCES-SUBDIR stormroot))
14                 url
15                 (do
16                    (log-message "Copying resources at " (str url) " to " target-dir)
17                    (FileUtils/copyDirectory (File. (.getFile url)) (File. target-dir))
18                    ))
19          )))
```

- 第2行的:local表明这是Local模式的实现，它的参数跟分布式模式的参数一样。
- 第3行创建存储该Topology信息的文件夹作为stormroot，它的路径是STORM-LOCAL-DIR/supervisor/stormdist/<storm-id>/。
- 第4行复制master-code-dir中的所有内容到stormroot下。
- 第5行获取当前线程的类加载器对象。
- 第6行调用resources-jar方法从java.class.path的系统参数中获取第一个包含resources文件夹的jar包。
- 第7行调用类加载器的getResource方法获取资源文件夹的地址。
- 第8行构造路径STORM-LOCAL-DIR/supervisor/stormdist/<storm-id>/resources/作为target-dir。
- 第9~18行判断怎样提取resources文件夹。如果第6行获取到jar包（不为空），那么就调用extract-dir-from-jar方法将resources文件夹提取出来并复制到STORM-LOCAL-DIR/supervisor/stormdist/<storm-id>/resources/目录下面。如果没有获得包含资源文件夹的jar包并且第7行获得的资源文件夹的地址不为空，就将该地址下对应的文件复制到目录中。这两种方式对应了在Local模式中以jar包方式运行和以文件夹方式运行。

第 9 章

Worker

9

Storm中的Worker是实际执行Topology的进程，它由Supervisor启动，从ZooKeeper中获取分配到自身的所有Executor并启动这些Executor来执行。

9.1 Worker 中的数据

Worker通过worker-data方法定义了一个包含很多共享数据的映射集合，这是一个很重要的数据结构，Worker中的很多方法都依赖它。表9-1详细描述了这个集合的信息。

表9-1 Worker数据

数据名称	获取方法	描　述
:conf	参数传入	配置项
:mq-context	msg-loader/mk-zmq-context	ZMQ的运行上下文
:storm-id	参数传入	StormID
:assignment-id	参数传入	任务分配id，即SupervisorId
:port	参数传入	Worker的端口号
:worker-id	参数传入	WorkerId
:cluster-state	cluster/mk-distributed-cluster-state conf	Cluster的状态，用于访问ZooKeeper
:storm-cluster-state	(cluster/mk-storm-cluster-state cluster-state)	Cluster中Topology的状态。通过访问ZooKeeper来获取cluster中Topology的元数据并将其存储于：:storm-cluster-state中
:storm-active-atom	(atom false)	Topology是否活跃的标志
:executors	read-worker-executors storm-conf storm-cluster-state storm-id assignment-id port	调用read-worker-executors来获得分配到该Worker上的所有Executor
:task-ids	(->> receive-queue-map keys (map int) sort)	Worker中含有的TaskId集合，根据Executor计算得到，后面将进一步讨论
:storm-conf	(read-supervisor-storm-conf conf storm-id)	获取Storm的配置项，后面将进一步讨论

（续）

数据名称	获取方法	描　　述
:topology	(read-supervisor-topology conf storm-id)	获取用户定义的Topology信息，后面将进一步讨论
:system-topology	(system-topology! storm-conf topology)	在用户定义的Topology的基础上添加系统组件，例如Acker Bolt等，后面将进一步讨论
:heartbeat-timer	(mk-halting-timer)	Worker心跳计时器
:refresh-connections-timer	(mk-halting-timer)	ZMQ连接维护计时器
:refresh-active-timer	(mk-halting-timer)	获取Topology状态的计时器
:executor-heartbeat-timer	(mk-halting-timer)	Executor心跳计时器
:user-timer	(mk-halting-timer)	Executor内部的定时器，Spout类型的Executor会定期向自身发送消息，然后该Executor根据收到的消息执行特定的操作，比如清理消息发送缓存等
:task-component	storm-task-info	TaskId到ComponentId的映射关系
:component->stream->fields	component->stream->fields	组件到流以及到流的所有字段的映射关系
:component->sorted-tasks	->> (:task->component <>) reverse-map (map-val sort)	Component到Task的映射关系
:endpoint-socket-lock	(mk-rw-lock)	用于更新ZMQ连接信息的锁
:cached-node+port->socket	atom {}	消息目标端node+port到ZMQ Socket缓存，详情可参见第5章
:cached-task->node+port	atom {}	taskId到其所在的node+port缓存，详情参见第5章
:transfer-queue	transfer-queue	Worker的消息发送队列。Executor向外发送消息时先将其放入该队列，再由Worker通过ZMQ发送出去
:executor-receive-queue-map	executor-receive-queue-map	Worker的消息接收队列集合，每个Executor对应其中一个队列
:short-executor-receive-queue-map	(map-key first executor-receive-queue-map)	StartTaskId到接收消息队列的映射关系
:task->short-executor	参数传入	TaskId到StartTaskId的映射关系
:suicide-fn	(mk-suicide-fn conf)	自杀函数
:uptime	(uptime-computer)	Worker的启动时间
:default-shared-resources	(mk-default-resources <>)	默认的共享资源
:user-shared-resources	(mk-user-resources <>)	用户资源，目前没有用到

（续）

数据名称	获取方法	描　　述
:transfer-local-fn	mk-transfer-local-fn	若目标Task位于同一 Worker时的消息发送方法，将直接发送至目标Task/Executor的接收消息队列，不需要串行化，具有较高效率
:transfer-fn	mk-transfer-fn	Worker的消息发送函数。当目标Task属于同一个Worker时，调用transfer-local-fn其他情况则发送:transfer-queue中的消息到ZMQ

9.2 Worker 中的计时器

每个计时器都对应着一个Java线程，Worker中使用计时器进行心跳保持以及获取元数据的更新信息。

9.2.1 Worker的心跳

do-heartbeat函数用于产生Worker的心跳信息，这些心跳信息被写入本地文件系统中。Supervisor会读取这些心跳信息以判断Worker的状态，然后决定是否需要重启Worker，相关代码如下：

```
(defn do-heartbeat [worker]
    (let [conf (:conf worker)
         hb (WorkerHeartbeat.
             (current-time-secs)
             (:storm-id worker)
             (:executors worker)
             (:port worker))]
    (log-debug "Doing heartbeat " (pr-str hb))
    ;; do the local-file-system heartbeat.
    (.put (worker-state conf (:worker-id worker))
        LS-WORKER-HEARTBEAT
        hb)
    ))
```

从该函数可以看出Worker的心跳包含如下信息.

- current-time-secs：当前时间。
- :storm-id：即TopologyId。
- :executors：Worker中包含的Executor列表。
- :port：与Worker对应的端口号。

worker-state方法会创建一个LocalState对象，并调用该对象的put方法将Worker的心跳信息存储到本地的文件系统中，对应路径是STORM-LOCAL-DIR/workers/<workerId>/heartbeats，存储

9

时使用的键为常量LS-WORKER-HEARTBEAT。Supervisor通过读取Worker的心跳信息来判断该Worker是否在正常运行。

图9-1展示了Worker的heartbeat目录下的文件，文件名为当前的时间戳。

1385390458287	11/25/2013 6:41 AM	File	2 KB
1385390458287.version	11/25/2013 6:41 AM	VERSION File	0 KB
1385390459288	11/25/2013 6:41 AM	File	2 KB
1385390459288.version	11/25/2013 6:41 AM	VERSION File	0 KB
1385390460290	11/25/2013 6:41 AM	File	2 KB
1385390460290.version	11/25/2013 6:41 AM	VERSION File	0 KB
1385390461294	11/25/2013 6:41 AM	File	2 KB
1385390461294.version	11/25/2013 6:41 AM	VERSION File	0 KB

图9-1　Worker的heartbeat目录

Storm采用:heartbeat-timer计时器来持续地发送心跳信息，每次发送的时间间隔由WORKER-HEARTBEAT-FREQUENCY-SECS来设定，默认值为1秒，相关代码如下：

```
_ (schedule-recurring (:heartbeat-timer worker) 0 (conf WORKER-HEARTBEAT-FREQUENCY-SECS)
    heartbeat-fn)
```

9.2.2　Executor的心跳

Worker的心跳与不同，Executor的心跳信息需要直接发送到ZooKeeper中保存。该心跳信息主要保存了Executor中Task的运行统计，Nimbus利用这些心跳信息判断Executor是否处于活跃状态并且还会在Storm UI上显示这些运行统计。

do-executor-heartbeats函数用来发送一次心跳信息，相关代码如下：

```
1 (defnk do-executor-heartbeats [worker :executors nil]
2    ;; stats is how we know what executors are assigned to this worker
3    (let [stats (if-not executors
4                    (into {} (map (fn [e] {e nil}) (:executors worker)))
5                    (->> executors
6                       (map (fn [e] {(executor/get-executor-id e) (executor/render-stats e)}))
7                       (apply merge)))
8            zk-hb {:storm-id (:storm-id worker)
9               :executor-stats stats
10              :uptime ((:uptime worker))
11              :time-secs (current-time-secs)
12              }]
13       ;; do the zookeeper heartbeat
14       (.worker-heartbeat! (:storm-cluster-state worker) (:storm-id worker) (:assignment-id
             worker) (:port worker) zk-hb)
15       ))
```

- 第6行通过executor/render-stats方法来获得Executor的运行统计信息，例如发送消息的数目等。

- 第8~12行构建Executor的心跳对象，包含如下信息。
  - :storm-id：即TopologyId。
  - :executor-stats：该Worker中Executor的运行统计，具体为对每一个Task的统计。
  - :uptime：Worker的启动时间。
  - :time-secs：当前时间。
- 第14行调用:storm-cluster-state的worker-heartbeat!方法存储心跳信息。在ZooKeeper中的默认路径为：

```
/storm/workerbeats/<storm-id> /<node-port>
```

同样地，Worker使用:executor-heartbeat-timer计时器线程来发送Executor的心跳信息，默认为3秒钟更新一次：

```
_ (schedule-recurring (:executor-heartbeat-timer worker) 0 (conf TASK-HEARTBEAT-FREQUENCY-SECS)
    #(do-executor-heartbeats worker :executors @executors))
```

9.2.3 Worker中对ZMQ连接的维护

在进程间，Storm利用ZMQ来发送和接收消息，并采用端到端的方式完成消息传输。Worker会根据Topology的定义以及分配到自身的任务情况，计算出自己发出的消息将被哪些Task接收。基于Topology的这一任务分配信息，Worker可以获悉目标Task所在的机器及端口号。虽然Worker会创建并缓存这些连接，但由于Worker上的分配任务可能被调整，因此Worker需要定时地更新这些连接信息。例如，若新增目标机器，则要增加连接并且关闭不需要的连接。

mk-refresh-connections函数用来更新这些ZMQ连接信息，相关代码如下：

```
1 (defn mk-refresh-connections [worker]
2   (let [outbound-tasks (worker-outbound-tasks worker)
3           conf (:conf worker)
4           storm-cluster-state (:storm-cluster-state worker)
5           storm-id (:storm-id worker)]
6       (fn this
7           ([]
8              (this (fn [& ignored] (schedule (:refresh-connections-timer worker) 0 this))))
97          ([callback]
10              (let [assignment (.assignment-info storm-cluster-state storm-id callback)
11                       my-assignment (-> assignment
12                                      :executor->node+port
13                                      to-task->node+port
14                                      (select-keys outbound-tasks)
15                                      (#(map-val endpoint->string %)))
16                   ;; we dont need a connection for the local tasks anymore
17                   needed-assignment (->> my-assignment
18                                          (filter-key (complement (-> worker :task-ids set))))
19                   needed-connections (-> needed-assignment vals set)
20                   needed-tasks (-> needed-assignment keys)
```

```
21
22                    current-connections (set (keys @(:cached-node+port->socket worker)))
23                    new-connections (set/difference needed-connections current-connections)
24                    remove-connections (set/difference current-connections needed-connections)]
25               (swap! (:cached-node+port->socket worker)
26                  #(HashMap. (merge (into {} %1) %2))
27                  (into {}
28                     (dofor [endpoint-str new-connections
29                                 :let [[node port] (string->endpoint endpoint-str)]]
30                          [endpoint-str
31                              (msg/connect
32                                 (:mq-context worker)
33                                 storm-id
34                                 ((:node->host assignment) node)
35                                 port)
36                              ]
37                          )))
38               (write-locked (:endpoint-socket-lock worker)
39                   (reset! (:cached-task->node+port worker)
40                       (HashMap. my-assignment)))
41               (doseq [endpoint remove-connections]
42                   (.close (get @(:cached-node+port->socket worker) endpoint)))
43               (apply swap!
44                   (:cached-node+port->socket worker)
45                   #(HashMap. (apply dissoc (into {} %1) %&))
46                   remove-connections)
47
48               (let [missing-tasks (->> needed-tasks
49                        (filter (complement my-assignment)))]
50                  (when-not (empty? missing-tasks)
51                     (log-warn "Missing assignment for following tasks: " (pr-str missing-tasks))
52                    )))))))
```

- 第2行通过调用`worker-outbound-tasks`函数得到`outbound-tasks`，该函数返回从Worker参数接收数据的TaskId集合。
- 第6行定义了`this`函数。在Clojure中，`this`只是普通的名字。
- 第7~8行定义函数的第一个重载，该重载定义了一个回调函数，它将在Worker的`:refresh-connections-timer`定时器上注册自己。
- 第9行开始定义一个带有`callback`参数的回调函数。
- 第10行调用`storm-cluster-state`的`assignment`函数获取与`storm-id`对应的Topology的任务分配。注意在将`callback`作为回调函数时，该函数会被注册到ZooKeeper的某个节点上充当Watcher函数，即当ZooKeeper中的节点发生变化时，客户端便会收到通知，此时更新Worker的Socket连接是非常好的时机。ZooKeeper中的Watcher回调函数在执行之后需要重新注册。
- 第11~15行获取接收该Worker消息的节点集合。`:executor->node+port`用来存储从Executor到node+port的映射关系；而`to-task->node+port`函数则会根据Executor中的TaskId集合来构建从TaskId到node+port的映射关系；接下来利用`select-keys`函数及`outbound-tasks`集合进行过滤，得到从该Worker接收消息的TaskId到node+port的映射关系；最后调用`map-val`

及endpoint->string函数获取node+port的字符串表示。最终结果为一个从TaskId到node+port的哈希表。

- 第17~18行对my-assignment进行过滤，移除所有属于该Worker的TaskId。这里需注意属于同一Worker的Task之间不需要利用ZMQ来完成通信。
- 在第19~20行中，needed-connections为目标Worker的节点集合，而needed-tasks则代表目标节点上的所有TaskId。
- 第22行获取在Worker节点上缓存的从node+port到ZMQ Socket的哈希表的所有键列表，并将其保存到current-connections变量中。
- 第23~24行判断哪些连接需要新建，哪些连接可以关闭。
- 第25~37行调用msg/connect方法，根据new-connections中的node+port创建新的连接，并放入集合current-connections中。
- 第38~40行将cached-task->node+port更新为my-assignment变量。
- 第41~46行调用需要删除的Socket的close方法，将这些Socket从:cached-node+port变量中移除。
- 第48~52行对异常情况进行处理，基本上不应该发生，若发生，可能会重新分配任务。例如，若属于当前Worker的任务被分配到其他Worker上，由于该函数会被计时器线程反复调用，故此处则只会打印警告信息。

最后来看一下这些函数的调用时机，相关代码如下：

```
1 :refresh-connections-timer (mk-halting-timer)
2 refresh-connections (mk-refresh-connections worker)
3 _ (refresh-connections nil)
4 (schedule-recurring (:refresh-connections-timer worker) 0 (conf TASK-REFRESH-POLL-SECS)
    refresh-connections)
```


- 第1行创建一个定时器。mk-halting-timer创建了一个计时器，该计时器会在异常发生时抛出异常并退出进程。该行代码会在创建Worker数据时调用。
- 第2~4行代码在mk-worker函数中被调用。第2行创建一个用于更新连接的函数，然后立即执行refresh-connections函数更新ZMQ消息。第4行将不断执行该函数，执行间隔为TASK-REFRESH-POLL-SECS，默认为10秒钟。

从代码中可以看出，Worker通过两种机制来保证连接的可靠性。一是在ZooKeeper中注册Watcher回调通知方法，这种方式并不一定可靠，例如若与ZooKeeper的连接丢失，则注册的Watcher回调方法将失效。二是采用定时器的方式来定期执行该函数。

9.2.4 从ZooKeeper获取Topology的活跃情况

Worker需要获知其执行的Topology的状态，例如，若用户已经把Topology的状态由活跃变为非活跃，Spout应停止向外发送消息。

refresh-storm-active函数用于获取Topology的状态信息，相关代码如下：

```
1 (defn refresh-storm-active
2   ([worker]
3       (refresh-storm-active worker (fn [& ignored] (schedule (:refresh-active-timer worker)
          0 (partial refresh-storm-active worker)))))
4   ([worker callback]
5       (let [base (.storm-base (:storm-cluster-state worker) (:storm-id worker) callback)]
6          (reset!
7             (:storm-active-atom worker)
8             (= :active (-> base :status :type))
9             ))
10       ))
```

- 第2~4行定义了refresh-storm-active的一个重载方法，它默认提供了一个匿名函数作为callback参数，该匿名函数被注册为ZooKeeper中getData方法的Watcher回调方法。当ZooKeeper中的数据发生变化时，该方法将被回调，于是Topology的状态就会在Worker中被及时更新。

  refresh-storm-active函数还会通过:refresh-active-timer计时器完成定期调用，默认的时间间隔为10秒钟，调用代码如下：

```
(schedule-recurring (:refresh-active-timer worker) 0 (conf TASK-REFRESH-POLL-SECS) (partial
    refresh-storm-active worker))
```

- 第5行调用:storm-cluster-state的storm-base方法获得Topology的基础信息。
- 第7行判断该Topology是否处于活跃状态（:active），并将判断结果存储于:storm-active-atom变量中。

9.2.5　小结

mk-halting-timer函数用于调用mk-timer函数来创建一个计时器，该计时器会在遇到错误时将错误记录到日志中并退出JVM，相关代码如下：

```
(defn mk-halting-timer []
    (mk-timer :kill-fn (fn [t]
        (log-error t "Error when processing event")
        (halt-process! 20 "Error when processing an event")
        )))
```

对Worker中用到的计时器的总结如表9-2所示。

表9-2　Worker中的计时器线程

计　时　器	回调方法	作　　用
:heartbeat-timer	do-heartbeat	Worker本地的心跳
:executor-heartbeat-timer	do-executor-heartbeats	Worker中Executor的心跳
:refresh-connections-timer	refresh-connections	更新ZMQ的连接信息
:refresh-active-timer	refresh-storm-active	判断Topology是否为活跃状态
:user-timer	在Spout类型的Executor中使用	在Executor中用于向SYSTEM_TICK_STREAM发送Tick消息

9.3 创建 Worker

mk-worker函数用于创建Worker进程，其主要工作包括启动相应的计时器、创建Worker中对应的Executor，以及启动接收线程来接收消息，相关代码如下：

```
1 (defserverfn mk-worker [conf shared-mq-context storm-id assignment-id port worker-id]
2     (log-message "Launching worker for " storm-id " on " assignment-id ":" port " with id
        " worker-id
3         " and conf " conf)
4     (if-not (local-mode? conf)
5         (redirect-stdio-to-slf4j!))
6     ;; because in local mode, its not a separate
7     ;; process. supervisor will register it in this case
8     (when (= :distributed (cluster-mode conf))
9         (touch (worker-pid-path conf worker-id (process-pid))))
10    (let [worker (worker-data conf shared-mq-context storm-id assignment-id port worker-id)
11            heartbeat-fn #(do-heartbeat worker)
12            ;; do this here so that the worker process dies if this fails
13            ;; it's important that worker heartbeat to supervisor ASAP when launching so
                that the supervisor knows it's running (and can move on)
14            _ (heartbeat-fn)
15
16            ;; heartbeat immediately to nimbus so that it knows that the worker has been started
17            _ (do-executor-heartbeats worker)
18
19
20            executors (atom nil)
21            ;; launch heartbeat threads immediately so that slow-loading tasks don't
                cause the worker to timeout
22            ;; to the supervisor
23            _ (schedule-recurring (:heartbeat-timer worker) 0 (conf WORKER-HEARTBEAT
                 -FREQUENCY-SECS) heartbeat-fn)
24            _ (schedule-recurring (:executor-heartbeat-timer worker) 0 (conf TASK-HEARTBEAT
                 -FREQUENCY-SECS) #(do-executor-heartbeats worker :executors @executors))
25
26
27            refresh-connections (mk-refresh-connections worker)
28
29            _ (refresh-connections nil)
30            _ (refresh-storm-active worker nil)
31
32            _ (reset! executors (dofor [e (:executors worker)] (executor/mk-executor worker e)))
33            receive-thread-shutdown (launch-receive-thread worker)
34
35            transfer-tuples (mk-transfer-tuples-handler worker)
36
37            transfer-thread (disruptor/consume-loop* (:transfer-queue worker) transfer-tuples)
38            shutdown* (单独讨论)
39            ret (reify
40                    Shutdownable
41                    (shutdown
42                        [this]
```

```
43                     (shutdown*))
44                 DaemonCommon
45                 (waiting? [this]
46                     (and
47                         (timer-waiting? (:heartbeat-timer worker))
48                         (timer-waiting? (:refresh-connections-timer worker))
49                         (timer-waiting? (:refresh-active-timer worker))
50                         (timer-waiting? (:executor-heartbeat-timer worker))
51                         (timer-waiting? (:user-timer worker))
52                         ))
53                 )]
54
55   (schedule-recurring (:refresh-connections-timer worker) 0 (conf TASK-REFRESH-POLL-SECS)
         refresh-connections)
56   (schedule-recurring (:refresh-active-timer worker) 0 (conf TASK-REFRESH-POLL-SECS) (partial
         refresh-storm-active worker))
57
58   (log-message "Worker has topology config " (:storm-conf worker))
59   (log-message "Worker " worker-id " for storm " storm-id " on " assignment-id ":" port "
         has finished loading")
60    ret
61    ))
```

- 在第4~5行代码中，若为分布式模式，则将打印到控制台的信息打到日志里面。
- 在第8~9行代码中，若为分布式模式，则将与Worker对应的进程ID放到pids目录下，并创建以进程ID作为文件名的空文件。Supervisor在关闭Worker时会尝试关闭pids目录下面所有与进程ID相对应的进程。Worker创建的子进程也应遵循这样的规则。在Storm中，由于任务会被重新调度，因此正在执行的Worker也可能被关闭。
- 第11~24行分别启动Worker以及Executor的心跳计时器线程。这里都是预先调用一次，以确保第一次的心跳信息可被快速发送出去，然后启动计时器线程来完成周期性的心跳更新。
- 第27~30行创建用于完成ZMQ连接更新的计时器线程。
- 第33行启动消息的接收线程，`receive-thread-shutdown`为该线程的关闭函数。
- 第35~37行启动消息队列的发送线程。

9.4　关闭 Worker

理解Worker的关闭函数有利于进一步理解Worker中启动的线程及资源，关闭Worker的函数的代码如下：

```
1 fn []
2     (log-message "Shutting down worker " storm-id " " assignment-id " " port)
3     (doseq [[_ socket] @(:cached-node+port->socket worker)]
4         ;; this will do best effort flushing since the linger period
5         ;; was set on creation
6         (.close socket))
```

```
7     (log-message "Shutting down receive thread")
8     (receive-thread-shutdown)
9     (log-message "Shut down receive thread")
10    (log-message "Terminating zmq context")
11    (log-message "Shutting down executors")
12    (doseq [executor @executors] (.shutdown executor))
13    (log-message "Shut down executors")
14
15    ;;this is fine because the only time this is shared is when it's a local context,
16    ;;in which case it's a noop
17    (msg/term (:mq-context worker))
18    (log-message "Shutting down transfer thread")
19    (disruptor/halt-with-interrupt! (:transfer-queue worker))
20
21    (.interrupt transfer-thread)
22    (.join transfer-thread)
23    (log-message "Shut down transfer thread")
24    (cancel-timer (:heartbeat-timer worker))
25    (cancel-timer (:refresh-connections-timer worker))
26    (cancel-timer (:refresh-active-timer worker))
27    (cancel-timer (:executor-heartbeat-timer worker))
28    (cancel-timer (:user-timer worker))
29
30    (close-resources worker)
31
32    ;; TODO: here need to invoke the "shutdown" method of WorkerHook
33
34    (.remove-worker-heartbeat! (:storm-cluster-state worker) storm-id assignment-id port)
35    (log-message "Disconnecting from storm cluster state context")
36    (.disconnect (:storm-cluster-state worker))
37    (.close (:cluster-state worker))
38    (log-message "Shut down worker " storm-id " " assignment-id " " port)
```

- 第2~4行关闭缓存的ZMQ的Socket连接。
- 第8行关闭消息接收线程。
- 第12行关闭Worker中的所有Executor线程。
- 第17行关闭ZMQ的上下文，释放已经创建的Socket连接。
- 第19~22行关闭消息发送队列和线程。
- 第24~28行关闭所有的计时器线程。
- 第30行关闭资源，目前尚未使用。
- 第34行从ZooKeeper中清除该Worker的心跳信息。
- 第36~37行用于断开与ZooKeeper的连接。

9.5 重要辅助方法介绍

在创建Worker中的数据结构、启动Worker以及关闭Worker的过程中会用到很多辅助方法，理解这些方法有助于我们更好地理解Worker的工作原理，下面简要介绍这些方法。

9.5.1　Worker中的接收函数

Worker中的mk-transfer-local-fn函数用于产生并发送消息到Executor的接收队列，同一Worker内部的Executor之间会通过该函数传递消息，其代码如下：

```
1 (defn mk-transfer-local-fn [worker]
2     (let [short-executor-receive-queue-map (:short-executor-receive-queue-map worker)
3               task->short-executor (:task->short-executor worker)
4               task-getter (comp #(get task->short-executor %) fast-first)]
5     (fn [tuple-batch]
6         (let [grouped (fast-group-by task-getter tuple-batch)]
7               (fast-map-iter [[short-executor pairs] grouped]
8                   (let [q (short-executor-receive-queue-map short-executor)]
9                      (if q
10                         (disruptor/publish q pairs)
11                         (log-warn "Received invalid messages for unknown tasks. Dropping... ")
12                         )))))))
```

- 第2行的short-executor-receive-queue-map存储Executor中第一个Task的TaskId到该Executor对应的接收队列（Disruptor Queue）的映射关系。
- 第3行的task->short-executor用于存储从该Worker中的TaskId到Executor中第一个Task的TaskId的映射关系。
- 第4行的task-getter函数以ZMQ发来的消息为传入参数。这里的消息为一个含有两个元素的数组，第一个元素为TaskId。task-getter函数的目标是通过消息的TaskId获得与其对应的Executor中第一个Task的TaskId。第二个元素为消息的实际内容。
- 第5~12行定义函数体，函数的输入为ZMQ收到的一组消息tuple-batch。第6行按照与消息TaskId对应的Executor中第一个Task的TaskId对消息进行分组，其变量grouped对应的键为Executor中第一个Task的TaskId，值为属于该Executor的一组消息。第8行通过Executor中第一个Task的TaskId获得与Executor相对应的接收消息队列q。第10行调用disruptor/publish方法将收到的消息发送至队列q中。若没有对应的接收队列，则将消息丢弃，这种情况是不应该发生的。

下面简单看一下fast-group-by的实现：

```
1 (defn fast-group-by [afn alist]
2   (let [ret (HashMap.)]
3      (fast-list-iter [e alist]
4          (let [key (afn e)
5                   ^List curr (get-with-default ret key (ArrayList.))]
6          (.add curr e)))
7      ret ))
```

该函数的传入参数为一个函数afn和一个列表alist。在当前的情况下，afn为task-getter函数，alist对应于接收到的消息。第2行定义哈希表类型的ret，用于存储分组的结果；第4行调用afn函数获得键，当前情况为获得TaskId；第6行将键对应的值初始化为一个列表。

9.5.2 Worker中的发送函数

Worker中的mk-transfer-fn函数与上一节中介绍的mk-transfer-local-fn类似，它产生的函数主要用于Executor的数据发送。这里存在两种情况，具体如下所示。

- 消息的目标TaskId跟发送TaskId属于同一个Worker，此时不需要跨进程传输消息，因此可将消息直接发送至接收端Executor的接收队列。
- 消息的目标TaskId跟发送TaskId属于不同的Worker，此时则将消息发送至Worker的发送队列，由Worker负责将队列中的消息通过ZMQ发送出去。

下面来看一下mk-transfer-fn函数的实现：

```
1 (defn mk-transfer-fn [worker]
2    (let [local-tasks (-> worker :task-ids set)
3              local-transfer (:transfer-local-fn worker)
4              ^DisruptorQueue transfer-queue (:transfer-queue worker)]
5         (fn [^KryoTupleSerializer serializer tuple-batch]
6             (let [local (ArrayList.)
7                       remote (ArrayList.)]
8                (fast-list-iter [[task tuple :as pair] tuple-batch]
9                    (if (local-tasks task)
10                       (.add local pair)
11                       (.add remote pair)
12                       ))
13                (local-transfer local)
14                ;; not using map because the lazy seq shows up in perf profiles
15                (let [serialized-pairs (fast-list-for [[task ^TupleImpl tuple] remote]
                          [task (.serialize serializer tuple)])]
16                   (disruptor/publish transfer-queue serialized-pairs)
17                   )))))
```

- 第2行的local-tasks为该Worker含有的TaskId集合。
- 第3行的local-transfer被设置为Worker中的:transfer-local-fn变量，该变量存储的是一个匿名函数，这个匿名函数通过调用9.5.1节中介绍的mk-transfer-local-fn函数得到。
- 第4行的transfer-queue为与Worker对应的消息发送队列。
- 第5行定义了一个函数，其参数为一个序列化器serializer和一组消息。
- 第6~12行将消息分组，local用于存储发送到同一个Worker中其他Task的消息，remote用于存储发送到其他Worker的Task的消息。
- 第13行调用local-transfer方法处理local列表中的消息。
- 第15行调用序列化器对消息进行序列化处理然后将其发送到transfer-queue中，注意这里只需要对消息进行序列化。

那么，Worker是如何接收数据的呢？Worker中会有一个额外的线程对transfer-queue进行监听，函数mk-transfer-tuples-hanlder用于创建与Disruptor Queue对应的消息处理器，相关代码如下：

```
1 (defn mk-transfer-tuples-handler [worker]
2    (let [^DisruptorQueue transfer-queue (:transfer-queue worker)
3              drainer (ArrayList.)
```

```
4            node+port->socket (:cached-node+port->socket worker)
5            task->node+port (:cached-task->node+port worker)
6            endpoint-socket-lock (:endpoint-socket-lock worker)
7            ]
8       (disruptor/clojure-handler
9          (fn [packets _ batch-end?]
10            (.addAll drainer packets)
11            (when batch-end?
12               (read-locked endpoint-socket-lock
13                  (let [node+port->socket @node+port->socket
14                           task->node+port @task->node+port]
15                     ;; consider doing some automatic batching here (would need to
                         not be serialized at this point to remove per-tuple overhead)
16                     ;; try using multipart messages ... first sort the tuples by the
                         target node (without changing the local ordering)
17
18                     (fast-list-iter [[task ser-tuple] drainer]
19                       ;; TODO: consider write a batch of tuples here to every
                               target worker
20                       ;; group by node+port, do multipart send
21                       (let [node-port (get task->node+port task)]
22                           (when node-port
23                               (msg/send (get node+port->socket node-port) task ser-tuple))
24                               ))))
25                  (.clear drainer))))))
```

- 第3行的drainer列表用于缓存要发送的消息。Disruptor Queue的Onevent回调会调用本函数定义的方法，该方法的最后一个参数表示Queue是否为一个Batch结束，并会在Batch结束之前将消息缓存到drainer列表中。
- 第4行的node+port->socket保存了Worker中与目标node+port相对应的ZMQ Socket连接。node+port代表Nimbus的资源分配单位，node则表示一台运行Supervisor的机器，port为该Supervisor上某一个运行Worker的端口号。
- 第5行的task->node+port为从TaskId到node+port的映射关系。
- 第6行的endpoint-socket-lock为Worker中定义的ReentrantReadWriteLock类型的锁，Worker中存在一个专门的线程，会对缓存的ZMQ连接进行更新。
- 第8行定义了一个clojure-handler，与其对应的函数在第9行定义。该函数的第1个传入参数为一组消息packets，第二个参数被忽略，第三个参数表明该Batch是否结束。
- 第10行将消息packets放入drainer变量中。若batch-end?为true，则为了避免跟ZMQ连接更新线程相冲突，这里需要申请读取endpoint-socket-lock锁，然后遍历drainer中缓存的所有消息，根据消息的taskId找到node+port，然后通过从node+port到ZMQSocket的映射关系找到对应的Socket连接。
- 第23行调用msg/send函数将消息发送出去。
- 第25行用于清理drainer缓存。

在Worker中，我们使用下面的方法来启动发送监听线程：

```
transfer-tuples (mk-transfer-tuples-handler worker)
transfer-thread (disruptor/consume-loop* (:transfer-queue worker) transfer-tuples)
```

9.5.3 获取属于Worker的Executor

read-worker-executors函数用来计算分配到该Worker的Executor，它通过调用storm-cluster-state的assignment-info函数获得所有Topology的分配信息，然后利用Worker的assignment-id以及port进行过滤，得到某个Worker所属的Executor，这里assignment-id对应于node。Worker启动后，其执行的Executor集合将不再发生变化。但当任务分配情况发生变化时，Supervisor就会重启Worker来处理任务。其中，Nimbus在计算任务分配时会尽量不改变Worker中已执行的Executor。当前Worker中任何一个Executor处理失败都会导致Worker重启，也即Worker中其他Executor的重新启动，Storm可以考虑对此进行优化。该函数的代码如下：

```
(defn read-worker-executors [storm-conf storm-cluster-state storm-id assignment-id port]
    (let [assignment (:executor->node+port (.assignment-info storm-cluster-state storm-id nil))]
      (doall
         (concat
            [Constants/SYSTEM_EXECUTOR_ID]
            (mapcat (fn [[executor loc]]
                        (if (= loc [assignment-id port])
                            [executor]
                            ))
                 assignment)))))
```

9.5.4 创建Executor的接收消息队列和查找表

mk-receive-queue-map函数用于为Worker中的每一个Executor创建接收队列，并将其存入哈希表，其中键为ExecutorId，值为Disruptor Queue的对象，其代码如下：

```
(defn- mk-receive-queue-map [storm-conf executors]
    (->> executors
         ;; TODO: this depends on the type of executor
         (map (fn [e] [e (disruptor/disruptor-queue (storm-conf TOPOLOGY-EXECUTOR-RECEIVE-
             BUFFER-SIZE)
                               :wait-strategy (storm-conf TOPOLOGY-DISRUPTOR-WAIT-STRATEGY))]))
          (into {})
         ))
```

ExecutorId实际上为含有两个元素的数组，即[startTaskId, endTaskId]，表示该Executor要执行的任务区间。

在Worker中，我们定义了几个哈希表以方便对信息进行查找。下面来看一下它们的关系：

```
1 executor-receive-queue-map (mk-receive-queue-map storm-conf executors)
2 receive-queue-map (->> executor-receive-queue-map
3     (mapcat (fn [[e queue]] (for [t (executor-id->tasks e)] [t queue])))
4     (into {}))
```

```
5
6 :task-ids (->> receive-queue-map keys (map int) sort)
7 :short-executor-receive-queue-map (map-key first executor-receive-queue-map)
8 :task->short-executor (->> executors
9    (mapcat (fn [e] (for [t (executor-id->tasks e)] [t (first e)])))
10   (into {})
11   (HashMap.))
12 :executor-receive-queue-map executor-receive-queue-map
```

- 第1行调用mk-receive-queue-map创建Disruptor Queue，并在第12行将其赋值给:executor-receive-queue-map。
- 第2~4行调用executor-id->tasks函数获得Executor中包含的TaskId集合，并创建哈希表，键为TaskId，值为该Task所属的Executor的接收队列。
- 第6行获得Executor中包含的TaskId集合，即为receive-queue-map的键集合。
- 第7行构建一个新的哈希表，存储从Executor的开始TaskId到该Executor的接收队列的映射关系。
- 第8~11行构建Executor中从TaskId到Executor的起始TaskId的映射关系。

于是，对于接收到的一组消息，根据其TaskId找到Executor的起始TaskId并根据其进行消息分组，然后根据从起始TaskId到Executor接收队列的哈希表short-executor-receive-queue-map来进行消息的分发。

在Worker中，TaskId、Executor以及Executor的收发队列的哈希表的关系如下所示。

- :executor-receive-queue-map：[startTaskId, endTaskId]->Executor接收队列；
- :short-executor-receive-queue-map：[startTaskId]->Executor接收队列；
- receive-queue-map：[taskId]->Executor接收队列；
- :task->short-executor：[taskId]->[startTaskId]。

9.5.5　下载Topology的配置项以及代码

在执行一个Topology时，Supervisor将从Nimbus下载3个文件，它们分别为stormconf.ser、stormcode.ser和stormjar.jar。其中，stormconf.ser为Topology配置项的序列化文件，read-supervisor-storm-conf函数用于读取该文件并将其反序列化。stormcode.ser为Topology的定义文件，可通过read-supervisor-topology函数来读取该文件，并将其反序列化解析。read-supervisor-storm-conf和read-supervisor-topology函数的代码如下：

```
(defn read-supervisor-storm-conf [conf storm-id]
    (let [stormroot (supervisor-stormdist-root conf storm-id)
            conf-path (supervisor-stormconf-path stormroot)
            topology-path (supervisor-stormcode-path stormroot)]
            (merge conf (Utils/deserialize (FileUtils/readFileToByteArray (File. conf-path))))
            ))


(defn read-supervisor-topology [conf storm-id]
    (let [stormroot (supervisor-stormdist-root conf storm-id)
```

```
        topology-path (supervisor-stormcode-path stormroot)]
    (Utils/deserialize (FileUtils/readFileToByteArray (File. topology-path)))
    ))
```

从Nimbus下载的这些文件在Supervisor所在机器的路径如下：

```
$StormRoot/stormData/supervisor/stormdist/<stormid>/
```

stormjar.jar文件包含了用户的资源文件，如第三方库等。Storm会将该jar文件进行解压，并放置于运行目录的resources文件夹下面，具体为：

```
$StormRoot/stormData/supervisor/stormdist/<stormid>/resources
```

9.6　小结

Worker中的线程及其通信关系如图9-2所示，读者可以结合代码进行分析。

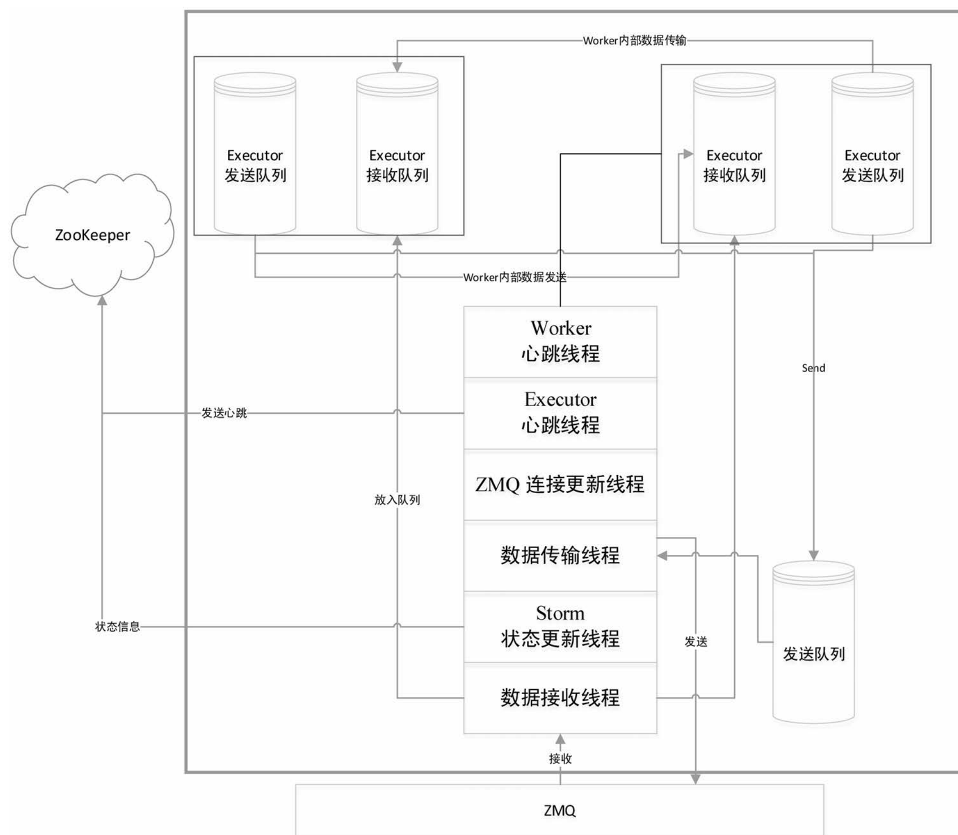


图9-2　Worker中的线程以及通信关系

第 10 章 Executor

Executor是Storm的核心运行单位，一个Executor实际上就是一个线程。一个Worker中可以启动一个或多个Executor，而一个Executor又可以含有多个Task，这些Task为逻辑运行单位属于相同的组件。一个 组件中含有的Executor数目与其并行度设置（parallelism_hint）有关，该参数表示组件希望以多少个线程来执行。每个组件还可以调用setNumTasks方法来设置其含有的Task数目为。Storm会根据一个组件的Task数目及并行度设置来计算哪些Task应该被分配到哪些Executor中，请参照第7章的内容来理解。Executor实际上包含了[startTaskId, endTaskId]区间内所有的Task。若未设置Task数目，则默认情况下Task数目与Executor数目相同，每个Executor中只含有个Task。

10.1 Executor 的数据

mk-executor-data函数用于定义Executor中含有的数据，理解Executor中的数据对于理解Executor是至关重要的。表10-1详细介绍了这些数据。

表10-1 Executor中的数据及其描述

数据名称	获取方法	描 述
:worker	参数传入	Executor所在的Worker引用
:worker-context	调用worker-context函数获得	WorkerTopologyContext对象
:executor-id	参数传入	Executor的标识符，为含有两个元素的数组[start-task-id, last-task-id]，它标识了Executor的任务区间
:task-ids	调用方法 executor-id->tasks获得	根据Executor的executor-id限定的TaskId的范围获得集合，返回与该Executor对应的TaskId
:component-id	.getComponentId worker-context (first task-ids)	与Executor对应的ComponentId， 通过调用WorkerTopologyContext对象的getComponentId方法获得
:open-or-prepare-was-called?	(atom false)	表征Spout的open方法或者Bolt的prepare方法是否被调用，与运行统计相关

（续）

数据名称	获取方法	描　　述
:storm-conf	normalized-component-conf (:storm-conf worker) worker-context component-id	将组件自定义的配置项与Topology的配置项合并，每个组件可以自定义如下配置项： ❑ TOPOLOGY-DEBUG ❑ TOPOLOGY-MAX-SPOUT-PENDING ❑ TOPOLOGY-MAX-TASK-PARALLELISM ❑ TOPOLOGY-TRANSACTIONAL-ID ❑ TOPOLOGY-TICK-TUPLE-FREQ-SECS ❑ TOPOLOGY-SLEEP-SPOUT-WAIT-STRATEGY-TIME-MS ❑ TOPOLOGY-SPOUT-WAIT-STRATEGY normalized-component-conf函数用于将配置项合并
:receive-queue	(:executor-receive-queue-map worker) executor-id	从Worker中获得该Executor的接收队列
:storm-id	(:storm-id worker)	从Worker中获得StormId标识符
:conf	(:conf worker)	获取Worker的配置
:shared-executor-data	(HashMap.)	表示一个Executor中所有Task的共享数据
:storm-active-atom	(:storm-active-atom worker)	表征该Topology是否为活跃状态，从Worker获取
:batch-transfer-queue	disruptor/disruptor-queue	Executor的消息发送队列，为Disruptor Queue类型
:transfer-fn	(mk-executor-transfer-fn batch-transfer->worker)	Executor的消息发送函数，通过mk-executor-transfer-fn函数获得，消息被发送到:batch-transfer-queue队列中
:suicide-fn	(:suicide-fn worker)	异常处理函数，用于退出进程，从Worker获得
:storm-cluster-state	cluster/mk-storm-cluster-state (:cluster-state worker)	Storm的运行状态，主要与ZooKeeper进行交互，其他章节有讨论
:stats	mk-executor-stats <> (sampling-rate storm-conf)	Executor的运行统计信息，Spout与Bolt对应于不同的统计信息。它通过调用mk-executor-stats定义
:type	executor-type worker-context component-id	Executor的类型，为Spout或者Bolt
:interval->task->metric-registry	(HashMap.)	系统的内置运行统计信息，含义为每个统计间隔上每个Task的统计信息
:task->component	(:task->component worker)	从Task到组件的对应关系，由Worker获得
:stream->component->grouper	outbound-components worker-context component-id	从流到接收组件及分组函数的哈希表
:report-error	throttled-report-error-fn <>	报告错误方法，后面会进一步讨论
:report-error-and-die	fn [error] ((:report-error) error) ((:suicide-fn))	报告错误并且退出
:deserializer	KryoTupleDeserializer. storm-conf worker-context	消息的反序列化器
:sampler	mk-stats-sampler storm-conf	运行统计采样器

10.2　Executor 的输入和输出

每一个Executor对应两个Disruptor Queue，分别用于输入与输出。

10.2.1　Executor的输入及处理

Worker在初始化时，会为其所包含的每一个Executor创建一个Disruptor Queue，用于接收数据。在创建Executor时，可根据其executor-id从Worker的:executor-receive-queue-map中获得该队列的引用：

```
:receive-queue ((:executor-receive-queue-map worker) executor-id)
```

当Worker的接收线程从ZMQ收到数据后，线程会根据目标的TaskId找到对应的Executor，并将数据发送到该Executor所对应的接收Disruptor Queue中。对于输入Disruptor Queue中的消息，Bolt类型的Executor会调用Bolt对象的Execute方法来处理，而Spout类型的Executor则调用Spout对象的Ack或Fail方法处理。

mk-task-receiver函数定义了一个函数来处理Disruptor Queue中的消息，它通过调用disruptor/clojure-handler函数获取一个消息处理函数，该消息处理函数会在收到新消息时被调用。clojure-handler函数要求传入一个函数。mk-task-receiver函数的代码如下：

```
1 (defn mk-task-receiver [executor-data tuple-action-fn]
2     (let [^KryoTupleDeserializer deserializer (:deserializer executor-data)
3             task-ids (:task-ids executor-data)
4             debug? (= true (-> executor-data :storm-conf (get TOPOLOGY-DEBUG)))
5             ]
6         (disruptor/clojure-handler
7             (fn [tuple-batch sequence-id end-of-batch?]
8                 (fast-list-iter [[task-id msg] tuple-batch]
9                     (let [^TupleImpl tuple (if (instance? Tuple msg) msg (.deserialize
                          deserializermsg))]
10                        (when debug? (log-message "Processing received message " tuple))
11                        (if task-id
12                            (tuple-action-fn task-id tuple)
13                            ;; null task ids are broadcast tuples
14                            (fast-list-iter [task-id task-ids]
15                                (tuple-action-fn task-id tuple)
16                                ))
17                        ))))))
```

- 第1行中形参tuple-action-fn是Executor处理消息的函数，而且Spout线程与Bolt线程处理消息的函数是不同的，这在后面会详细介绍。
- 收到消息后，将调用Disruptor的onEvent函数，第7~16行定义该函数。在第9行中，若消息已经为Tuple对象，则不需要再进行反序列化。例如，该消息是由属于同一个Worker的其他Task发送过来的。

- 第11行，若存在消息的来源task-id，则调用一次tuple-action-fn函数；若未指定task-id，则在该Executor中的所有Task上调用tuple-action-fn函数。

在创建Spout或Bolt时，会调用mk-task-receiver函数并将结果存储于event-handler变量中：

```
event-handler (mk-task-receiver executor-data tuple-action-fn)
```

在Spout中以非阻塞方式接收数据：

```
(disruptor/consume-batch receive-queue event-handler)
```

而在Bolt中，则以阻塞方式接收数据：

```
(disruptor/consume-batch-when-available receive-queue event-handler)
```

Bolt的消息循环主要调用event-handler中的方法来对消息进行处理。其实Bolt的数据接收并不是严格意义上阻塞的，即若等待一定时间后仍然没有输入，函数将返回空集合。

10.2.2　Executor的输出及发送

每个Executor都会产生一个用于输出的Disruptor Queue对象，Executor在发送消息时首先会将消息内容发送到该队列中。Executor会启动一个发送线程来处理该队列中的数据，该线程调用Worker中由mk-transfer-fn产生的函数对数据进行处理，或者把数据通过ZMQ发送到其他Worker，或者直接发送到与该Worker上的其他Executor相对应的接收Disruptor Queue中。

start-batch-transfer->worker-handler! 函数会调用disruptor/consume-loop*函数来启动用于发送数据队列的发送线程，相关代码如下：

```
1 (defn start-batch-transfer->worker-handler! [worker executor-data]
2     (let [worker-transfer-fn (:transfer-fn worker)
3              cached-emit (MutableObject. (ArrayList.))
4              storm-conf (:storm-conf executor-data)
5              serializer (KryoTupleSerializer. storm-conf (:worker-context executor-data))
6              ]
7         (disruptor/consume-loop*
8             (:batch-transfer-queue executor-data)
9             (disruptor/handler [o seq-id batch-end?]
10                (let [^ArrayList alist (.getObject cached-emit)]
11                    (.add alist o)
12                    (when batch-end?
13                        (worker-transfer-fn serializer alist)
14                        (.setObject cached-emit (ArrayList.))
15                        )))
16            :kill-fn (:report-error-and-die executor-data))))
```

- 第2行的worker-transfer-fn为Worker中由mk-transfer-fn定义的函数，用于发送Disruptor Queue中的一个数据。
- 第3行的cached-emit为列表类型的对象数组，用于缓存要发送的数据。Disruptor Queue是

批量处理数据的，它在真正处理这批数据之前，数据会被放入cached-emit列表中。

- 第5行生成一个KryoTupleSerializer，用来对发送的数据进行序列化。
- 第7~15行启动Disruptor Queue的消费者线程。第8行的：batch-transfer-queue为Executor定义的Disruptor Queue对象。在第12~15行中，batch-end?参数用于表明Disruptor Queue是否准备对消息进行处理。若处理，则调用worker-transfer-fn发送缓存的消息，同时将cached-emit对象重置为一个空列表。
- 第16行传入Executor的report-error-and-die函数，该函数将对错误进行记录并退出进程。

在创建Executor的过程中，我们会启动线程system-threads（mk-executor函数）：

```
system-threads [(start-batch-transfer->worker-handler! worker executor-data)]
```

Executor在创建其数据时会创建该发送队列，下面的代码用于创建发送队列：

```
batch-transfer->worker (disruptor/disruptor-queue
    (storm-conf TOPOLOGY-EXECUTOR-SEND-BUFFER-SIZE)
    :claim-strategy :single-threaded
    :wait-strategy (storm-conf TOPOLOGY-DISRUPTOR-WAIT-STRATEGY))
```

10.3　Spout 类型的 Executor

mk-threads函数用于创建与Executor对应的消息循环主线程。Spout和Bolt有着不同的消息循环策略，故需分别进行定义。下面的代码用于定义接口，Spout和Bolt会分别实现该接口：

```
(defmulti mk-threads executor-selector)
```

mk-threads函数的主消息循环通过async-loop方法实现，若传入的函数为工厂方法，则在第一次调用该方法时进行初始化，并返回用于消息循环的函数。为了便于讨论以及节约篇幅，本节将前面讨论过的方法实现略去。

10.3.1　准备消息循环的数据

这部分数据在mk-threads的let方法中定义：

```
1 (let [{:keys [storm-conf component-id worker-context transfer-fn report-error sampler
     open-or-prepare-was-called?]} executor-data
2          ^ISpoutWaitStrategy spout-wait-strategy (init-spout-wait-strategy storm-conf)
3          max-spout-pending (executor-max-spout-pending storm-conf (count task-datas))
4          ^Integer max-spout-pending (if max-spout-pending (int max-spout-pending))
5          last-active (atom false)
6          spouts (ArrayList. (map :object (vals task-datas)))
7          rand (Random. (Utils/secureRandomLong))
8
9          pending (RotatingMap.2
10                   ;; microoptimize for performance of .size method
```

```
11                 (reify RotatingMap$ExpiredCallback
12                   (expire [this msg-id [task-id spout-id tuple-info start-time-ms]]
13                     (let [time-delta (if start-time-ms (time-delta-ms start-time-ms))]
14                       (fail-spout-msg executor-data (get task-datas task-id)
                         spout-id tuple-info time-delta)
15                       ))))
16           tuple-action-fn (后面会单独介绍)
17           receive-queue (:receive-queue executor-data)
18           event-handler (mk-task-receiver executor-data tuple-action-fn)
19           has-ackers? (has-ackers? storm-conf)
20           emitted-count (MutableLong. 0)
21           empty-emit-streak (MutableLong. 0)
22           overflow-buffer (LinkedList.)]
```

- 第1行表明了mk-threads函数将使用的Executor数据。
- 第2行定义Spout的等待策略spout-wait-strategy，该策略通过调用init-spout-wait-strategy函数来得到，由配置项TOPOLOGY-SPOUT-WAIT-STRATEGY进行配置。该策略用于Spout的nextTuple函数多次没有消息输出的情况，此时系统将默认使用SleepSpoutWaitStrategy，即该情况下Spout将睡眠一段时间。
- 第3行获得消息的max-spout-pending数目，通过计算配置项TOPOLOGY-MAX-SPOUT-PENDING与当前Executor中Task数目的乘积来获得。当该Executor中存在超过max-spout-pending数目的消息没有被Ack或Fail时，Spout将进入等待状态，并利用该参数进行流量控制。
- 第5行的last-active表明Spout的状态。当Topology处于非活跃状态时，last-active将被设置为false，此时Spout不会向外发送数据。
- 第6行获得该Executor包含的Spout对象集合。每一个Task对应着一个Spout对象，依次调用Spout的nextTuple方法来发送消息。
- 第9~15行定义pending变量，用于存储已发送出去但未被Ack或Fail的消息。在RotatingMap对象中，我们可以设置回调函数。当系统收到SYSTEM_TICK的时候，将调用pending的rotate方法。于是第11~15行定义的ExpiredCallback方法会被调用，以对每一条消息调用fail-spout-msg函数进行处理，即调用Spout的Fail回调。
- 第16行的tuple-action-fn函数用于处理Spout收到的消息。
- 第17行的receive-queue对应于Executor的接收队列。
- 第18行的event-handler用来处理Spout收到的消息，它将调用tuple-action-fn函数。
- 第19行定义has-ackers?来表明系统中是否存在Acker Bolt。
- 第22行定义了overflow-buffer，在Executor所对应的Disruptor Queue发送队列已被填满时，消息将被发送到overflow-buffer中。Spout发送消息时，优先发送overflow-buffer中的数据。另外，Spout的主循环要求每一步操作都是非阻塞的。
- 第20行定义的emitted-count用来记录Executor发送的消息数目，而第21行empty-emit-streak变量则用来记录连续调用nextTuple函数且无消息发送的数目，它会被当作spout-wait-strategy的emitEmpty方法的参数。

10.3.2　Spout输入处理函数

对于Spout类型的Executor来讲，输入消息主要是系统消息，例如对发出消息进行回复的Ack/Fail消息、系统的tick消息等。Spout主线程消息循环需要做很多工作，为了不影响其他工作，Spout会采用非阻塞的方式从接收队列中获取消息。

Spout在收到消息后调用的消息处理函数如下：

```
1 tuple-action-fn (fn [task-id ^TupleImpl tuple]
2     (let [stream-id (.getSourceStreamId tuple)]
3         (condp = stream-id
4             Constants/SYSTEM_TICK_STREAM_ID (.rotate pending)
5             Constants/METRICS_TICK_STREAM_ID (metrics-tick executor-data
                  task-datas tuple)
6             (let [id (.getValue tuple 0)
7                 [stored-task-id spout-id tuple-finished-info start-time-ms]
                      (.remove pending id)]
8                 (when spout-id
9                     (when-not (= stored-task-id task-id)
10                        (throw-runtime "Fatal error, mismatched task ids: "task-id " "
                             stored-task-id))
11            (let [time-delta (if start-time-ms (time-delta-msstart-time-ms))]
12                (condp = stream-id
13                   ACKER-ACK-STREAM-ID (ack-spout-msg executor-data (get
                         task-datas task-id)
14                       spout-id tuple-finished-info time-delta)
15                   ACKER-FAIL-STREAM-ID (fail-spout-msg executor-data
                         (get task-datas task-id)
16                       pout-id tuple-finished-info time-delta)
17                       )))
18        ;; TODO: on failure, emit tuple to failure stream
19        ))))
```

第2行计算收到消息的来源。若消息来自`SYSTEM_TICK_STREAM-ID`，则调用pending对象的`rotate`方法，该方法将导致发送消息超时。在进行消息跟踪的过程中，Spout会用pending对象来保存所有发送出去的消息，用`SYSTEM_TICK`消息作为清理缓存消息的信号。例如，若某个消息的Ack消息丢失，导致Spout中的该消息没有被Ack，则Spout会在收到Tick消息后调用pending对象的超时操作来清理缓存，然后调用该消息的Fail操作并交由用户来决定是进行重传还是将其忽略。若消息来自`METRICS_TICK_STREAM_ID`，则调用`metrics-tick`方法来整理目前的统计信息并将其发送到信息统计Bolt节点上去。

其他的消息来源只能为Ack/Fail的流。于是，第6行获得消息的第1列，即发送的消息ID，它是一个在发送时随机产生的long类型的对象。第7行将该ID对应的数据从pending数组中清除出去，该ID对应的数据被返回，其内容为：

```
[stored-task-id, spout-id, tuple-finished-info, start-time-ms]
```

其中`stored-task-id`为发送该消息的TaskId，`spout-id`为发送消息时附带的MessageId，

tuple-finished-info含有发送消息的StreamId以及消息内容，start-time-ms则在消息被执行统计采样时，存储为当前时间，否则为空，如下面的代码所示：

```
(.put pending root-id [task-id
    message-id
    {:stream out-stream-id :values values}
    (if (sampler) (System/currentTimeMillis))])
```

若消息来自于ACKER-ACK-STREAM-ID，则调用ack-spout-msg回调方法处理消息；若消息来自ACKER-FAIL-STREAM-ID，则调用fail -spout -msg方法进行处理。

ack-spout-msg函数主要调用用户的Spout对象的Ack回调方法，同时更新相关的统计信息，其代码如下：

```
(defn- ack-spout-msg [executor-data task-data msg-id tuple-info time-delta]
    (let [storm-conf (:storm-conf executor-data)
            ^ISpout spout (:object task-data)
            task-id (:task-id task-data)]
        (when (= true (storm-conf TOPOLOGY-DEBUG))
            (log-message "Acking message " msg-id))
        (.ack spout msg-id)
        (task/apply-hooks (:user-context task-data) .spoutAck (SpoutAckInfo. msg-id task-id
            time-delta))
         (when time-delta
             (builtin-metrics/spout-acked-tuple! (:builtin-metrics task-data) (:stats
                 executor-data) (:stream tuple-info) time-delta)
             (stats/spout-acked-tuple! (:stats executor-data) (:stream tuple-info)
                 time-delta))))
```

fail-spout-msg函数主要调用Spout对象的Fail回调方法，其代码如下：

```
(defn- fail-spout-msg [executor-data task-data msg-id tuple-info time-delta]
    (let [^ISpout spout (:object task-data)
            task-id (:task-id task-data)]
        ;;TODO: need to throttle these when there's lots of failures
        (log-debug "Failing message " msg-id ": " tuple-info)
        (.fail spout msg-id)
        (task/apply-hooks (:user-context task-data) .spoutFail (SpoutFailInfo. msg-id task-id
            time-delta))
        (when time-delta
            (builtin-metrics/spout-failed-tuple! (:builtin-metrics task-data) (:stats executor-data)
               (:stream tuple-info))
            (stats/spout-failed-tuple! (:stats executor-data) (:stream tuple-info) time-delta))))
```


注意到Spout的fail方法只是传入了MessageId对象，作者认为既然已经在pending发送队列中保存了消息内容，为什么不将该消息直接通过Fail方法返回给用户的Spout呢？这样用户收到失败消息后便可更方便地进行重传。

10.3.3　Spout消息发送函数

Spout使用send-spout-msg函数来发送消息，其代码如下：

```
1 send-spout-msg (fn [out-stream-id values message-id out-task-id]
2                   (.increment emitted-count)
3                   (let [out-tasks (if out-task-id
4                                     (tasks-fn out-task-id out-stream-id values)
5                                     (tasks-fn out-stream-id values))
6                         rooted? (and message-id has-ackers?)
7                         root-id (if rooted? (MessageId/generateId rand))
8                         out-ids (fast-list-for [t out-tasks] (if rooted?
                              (MessageId/generateId rand)))]
9                     (fast-list-iter [out-task out-tasks id out-ids]
10                    (let [tuple-id (if rooted?
11                                     (MessageId/makeRootId root-id id)
12                                     (MessageId/makeUnanchored))
13                              out-tuple (TupleImpl. worker-context
14                                            values
15                                            task-id
16                                            out-stream-id
17                                            tuple-id)]
18                        (transfer-fn out-task
19                           out-tuple
20                           overflow-buffer)
21                        ))
22                    (if rooted?
23                       (do
24                            (.put pending root-id [task-id
25                                                   message-id
26                                                   {:stream out-stream-id :values values}
27                                                 (if (sampler) (System/currentTimeMillis))])
28                            (task/send-unanchored task-data
29                               ACKER-INIT-STREAM-ID
30                               [root-id (bit-xor-vals out-ids) task-id]
31                               overflow-buffer))
32                       (when message-id
33                           (ack-spout-msg executor-data task-data message-id
34                              {:stream out-stream-id :values values}
35                              (if (sampler) 0))))
36                    (or out-tasks [])
37                    ))]]
```

在该函数中，其中各个形参的含义为：out-stream-id是消息的StreamId；values是消息内容；message-id是消息的MessageId，表示是否要对消息进行跟踪；out-task-id则是消息的接收端TaskId，用于向直接流发送消息。

- 第3~5行调用tasks-fn函数来获得消息的目标TaskId。tasks-fn是Task的主要函数，它会根据消息的StreamId和消息内容来确定哪些Task将接收该流的消息，以及以何种方式来接收该流的消息。对于直接分组方式，其作用主要是检查目标out-task-id是否以直接分组的方式来接收消息。

- 第18~21行调用transfer-fn函数来发送消息。该函数由mk-executor-transfer-fn函数创建，并会将消息发送至Executor的发送队列中。

其他的代码与消息跟踪相关，请参看第12章。

前面提到的mk-executor-transfer-fn函数用于定义消息发送函数，其参数batch-transfer->worker对应于Executor的输出Disruptor Queue，其代码如下：

```
1 (defn mk-executor-transfer-fn [batch-transfer->worker]
2    (fn this
3       ([task tuple block? ^List overflow-buffer]
4           (if (and overflow-buffer (not (.isEmpty overflow-buffer)))
5              (.add overflow-buffer [task tuple])
6              (try-cause
7                 (disruptor/publish batch-transfer->worker [task tuple] block?)
8                 (catch InsufficientCapacityException e
9                    (if overflow-buffer
10                       (.add overflow-buffer [task tuple])
11                       (throw e))
12                   ))))
13      ([task tuple overflow-buffer]
14          (this task tuple (nil? overflow-buffer) overflow-buffer))
15      ([task tuple]
16         (this task tuple nil)
17         )))
```

该函数存在3种重载，主要区别在于是否对发送的消息进行缓存。

- 在第4行中，若overflow-buffer非空，则将消息放入其中；否则通过disruptor/publish方法进行发送。
- 从第8~12行可以看出，当Disruptor Queue空间不足时，函数会将要发送的消息放入overflow-buffer中。overflow-buffer是一个临时缓存，当Disruptor Queue接收端未启动或空间不足时，用于临时存放将要发送的消息。第4行的overflow-buffer非空，则表明该异常已经发生过，Disruptor Queue中空间已经不足，此时消息会被直接放入overflow-buffer以提高效率。在Spout的消息循环中，会优先发送overflow-buffer中的数据。

10.3.4 Spout对象的初始化

下面的代码用于调用Executor中各Spout对象的open操作，其中open方法只会被调用一次：

```
1  ;; If topology was started in inactive state, don't call (.open spout) until it's activated first.
2 (while (not @(:storm-active-atom executor-data))
3   (Thread/sleep 100))
4
5 (log-message "Opening spout " component-id ":" (keys task-datas))
6 (doseq [[task-id task-data] task-datas
7           :let [^ISpout spout-obj (:object task-data)
8                 tasks-fn (:tasks-fn task-data)
9                 send-spout-msg (参考前面讨论)]]
```

```
10     (builtin-metrics/register-all (:builtin-metrics task-data) storm-conf (:user-context
           task-data))
11     (.open spout-obj
12         storm-conf
13         (:user-context task-data)
14         (SpoutOutputCollector.
15            (reify ISpoutOutputCollector
16               (^List emit [this ^String stream-id ^List tuple ^Object message-id]
17                  (send-spout-msg stream-id tuple message-id nil)
18                  )
19               (^void emitDirect [this ^int out-task-id ^String stream-id
20                                     ^List tuple ^Object message-id]
21                  (send-spout-msg stream-id tuple message-id out-task-id)
22                  )
23               (reportError [this error]
24                  (report-error error)
25                  )))))
26 (reset! open-or-prepare-was-called? true)
27 (log-message "Opened spout " component-id ":" (keys task-datas))
28 (setup-metrics! executor-data)
29
30 (disruptor/consumer-started! (:receive-queue executor-data))
```

- 第2~3行进行等待直到Topology处于活跃状态。
- 第6~25行对Executor中的每一个Spout进行操作。第8~9行获得`tasks-fn`函数和`send-spout-msg`函数。`send-spout-msg`函数会利用`tasks-fn`函数来选择目标TaskId。注意到Executor中的每一个Spout都定义了`send-spout-msg`方法，这是由于消息接收端TaskId可能会与当前Task相关。例如，直接分组方式中，消息接收端的TaskId以及当前Task是预先绑定的。类似地，`send-spout-msg`还需要利用`tasks-fn`来选择接收端TaskId，同时在进行消息跟踪时记录发送消息的TaskId，故Storm将这些函数定义为Task级别而非Executor共享。
- 第11~25行依次调用Spout对象的`open`回调方法，同时实例化`SpoutOutputCollector`，它主要调用`send-spout-msg`来发送消息。
- 第30行调用`consumer-started!`函数打开接收队列。由于在`open`函数被调用之前，接收队列尚未打开，故最好不要在Spout的`open`函数中发送消息（当然即使真的这样发送了Storm也会对此时发送的消息进行缓存处理）。

10.3.5　消息循环

传入`async-loop`的工厂函数被调用后，会返回第1~39行定义的函数。在`async-loop`循环体中，将调用该函数，从第39行该函数的返回值为0可以看出，相邻两次函数调用之间是没有等待的。正常情况下，该函数会依次执行接收队列的消息、重发之前未成功发送的消息、调用`nextTuple`发送新的消息，以及进行流量控制等一系列操作。下面来看这个函数的实现：

```
1 (fn []
2   ;; This design requires that spouts be non-blocking
3   (disruptor/consume-batch receive-queue event-handler)
4
5   ;; try to clear the overflow-buffer
6   (try-cause
7      (while (not (.isEmpty overflow-buffer))
8          (let [[out-task out-tuple] (.peek overflow-buffer)]
9              (transfer-fn out-task out-tuple false nil)
10             (.removeFirst overflow-buffer)))
11     (catch InsufficientCapacityException e
12        ))
13
14     (let [active? @(:storm-active-atom executor-data)
15             curr-count (.get emitted-count)]
16         (if (and (.isEmpty overflow-buffer)
17                  (or (not max-spout-pending)
18                     (< (.size pending) max-spout-pending)))
19            (if active?
20               (do
21                   (when-not @last-active
22                        (reset! last-active true)
23                        (log-message "Activating spout " component-id ":" (keys task-datas))
24                        (fast-list-iter [^ISpout spout spouts] (.activate spout)))
25
26                   (fast-list-iter [^ISpout spout spouts] (.nextTuple spout)))
27               (do
28                   (when @last-active
29                       (reset! last-active false)
30                       (log-message "Deactivating spout " component-id ":" (keys task-datas))
31                       (fast-list-iter [^ISpout spout spouts] (.deactivate spout)))
32                   ;; TODO: log that it's getting throttled
33                   (Time/sleep 100))))
34         (if (and (= curr-count (.get emitted-count)) active?)
35            (do (.increment empty-emit-streak)
36                  (.emptyEmit spout-wait-strategy (.get empty-emit-streak)))
37            (.set empty-emit-streak 0)
38            ))
39     0))
```

- 第3行以非阻塞的方式对接收队列中的消息进行处理。
- 第5~12行优先发送overflow-buffer中的数据（由于Executor中发送消息队列已满，数据会被缓存在overflow-buffer中）。
- 在第16~18行中，若overflow-buffer为空，并且pending存储的数据少于max-spout-pending，或者未设置max-spout-pending，最后需要Topology处于活跃状态，则Spout可以发送消息。
- 第26行依次调用spout的nextTuple回调方法来发送消息。nextTuple会利用传入的SpoutOutputCollector的emit或emitDirect方法来发送消息，并最终调用send-spout-msg函数将消息发送到Executor的消息发送队列中。send-spout-msg函数会更新emitted-count。
- 在第28~33行中，若Topology处于非活跃状态，则睡眠100毫秒。

10

- 在第34~38行中，若emitted-count与上次发送消息的curr-count相同，则表明nextTuple函数没有发送出去消息，此时调用spout-wait-strategy的emptyEmit方法来进行处理。默认情况下，我们会根据TOPOLOGY_SLEEP_SPOUT_WAIT_STRATEGY_TIME_MS配置项来决定睡眠时间，目前为1毫秒。

10.4　Bolt 类型的 Executor

Bolt类型的Executor的定义跟Spout类型的类似，下面我们来详细介绍一下。

10.4.1　准备消息循环的数据

Bolt类型的Executor的消息循环数据较少，主要是定义了tuple-action-fn函数（后面会单独介绍），该函数会根据TaskId获得对应的Bolt对象并调用其execute方法。相关代码如下：

```
(let [execute-sampler (mk-stats-sampler (:storm-conf executor-data))
    executor-stats (:stats executor-data)
    {:keys [storm-conf component-id worker-context transfer-fn report-error sampler
            open-or-prepare-was-called?]} executor-data
    rand (Random. (Utils/secureRandomLong))
    tuple-action-fn],
```

10.4.2　Bolt输入处理函数

Bolt的消息处理较为简单，相关代码如下所示：

```
1 tuple-action-fn (fn [task-id ^TupleImpl tuple]
2   (let [stream-id (.getSourceStreamId tuple)]
3       (condp = stream-id
4           Constants/METRICS_TICK_STREAM_ID (metrics-tick executor-data task-datas tuple)
5           (let [task-data (get task-datas task-id)
6                     ^IBolt bolt-obj (:object task-data)
7                     user-context (:user-context task-data)
8                     sampler? (sampler)
9                     execute-sampler? (execute-sampler)
10                    now (if (or sampler? execute-sampler?) (System/currentTimeMillis))]
11              (when sampler?
12                (.setProcessSampleStartTime tuple now))
13              (when execute-sampler?
14                (.setExecuteSampleStartTime tuple now))
15              (.execute bolt-obj tuple)
16              (let [delta (tuple-execute-time-delta! tuple)]
17                  (task/apply-hooks user-context .boltExecute (BoltExecuteInfo. tuple task-id
                        delta))
18                  (when delta
19                      (builtin-metrics/bolt-execute-tuple! (:builtin-metrics task-data)
20                          executor-stats
21                          (.getSourceComponent tuple)
```

```
22                             (.getSourceStreamId tuple)
23                             delta)
24                         (stats/bolt-execute-tuple! executor-stats
25                             (.getSourceComponent tuple)
26                             (.getSourceStreamId tuple)
27                             delta)))))))]
```

第6行获得该Bolt对应的bolt-obj，第15行调用bolt-obj的execute回调方法。其他处理逻辑都与运行统计相关，请参考第13章和第14章。

10.4.3 Bolt的消息发送函数

Bolt使用bolt-emit函数来发送消息，其代码如下：

```
1 bolt-emit (fn [stream anchors values task]
2   (let [out-tasks (if task
3                     (tasks-fn task stream values)
4                     (tasks-fn stream values))]
5       (fast-list-iter [t out-tasks]
6          (let [anchors-to-ids (HashMap.)]
7              (fast-list-iter [^TupleImpl a anchors]
8                  (let [root-ids (-> a .getMessageId .getAnchorsToIds .keySet)]
9                      (when (pos? (count root-ids))
10                         (let [edge-id (MessageId/generateId rand)]
11                             (.updateAckVal a edge-id)
12                             (fast-list-iter [root-id root-ids]
13                                 (put-xor! anchors-to-ids root-id edge-id))
14                             ))))
15             (transfer-fn t
16                        (TupleImpl. worker-context
17                            values
18                            task-id
19                            stream
20                            (MessageId/makeId anchors-to-ids)))))
21       (or out-tasks [])))
```

第2~4行获取消息接收端的TaskId集合，第15~20行调用transfer-fn函数发送消息，该函数与Spout中的一致（差别在于Bolt不使用overflow-buffer缓存）。其他的代码与消息跟踪相关，详情请参考第12章。

10.4.4 Bolt对象的初始化

以下这部分代码主要用于调用Bolt的prepare函数：

```
1 ;; If topology was started in inactive state, don't call prepare bolt until it's activated first.
2  (while (not @(:storm-active-atom executor-data))
3      (Thread/sleep 100))
4
5  (log-message "Preparing bolt " component-id ":" (keys task-datas))
6  (doseq [[task-id task-data] task-datas
```

```
7            :let [^IBolt bolt-obj (:object task-data)
8                  tasks-fn (:tasks-fn task-data)
9                  user-context (:user-context task-data)
10                 bolt-emit]]
11    (builtin-metrics/register-all (:builtin-metrics task-data) storm-conf user-context)
12    (.prepare bolt-obj
13       storm-conf
14       user-context
15       (OutputCollector.
16          (reify IOutputCollector
17              (emit [this stream anchors values]
18                 (bolt-emit stream anchors values nil))
19              (emitDirect [this task stream anchors values]
20                 (bolt-emit stream anchors values task))
21              (^void ack [this ^Tuple tuple]
22                 (let [^TupleImpl tuple tuple
23                          ack-val (.getAckVal tuple)]
24                     (fast-map-iter [[root id] (.. tuple getMessageId getAnchorsToIds)]
25                         (task/send-unanchored task-data
26                             ACKER-ACK-STREAM-ID
27                             [root (bit-xor id ack-val)])
28                             )))
29              (^void fail [this ^Tuple tuple]
30                  (fast-list-iter [root (.. tuple getMessageId getAnchors)]
31                      (task/send-unanchored task-data
32                          ACKER-FAIL-STREAM-ID
33                          [root])))
34              (reportError [this error]
35                  (report-error error)
36                  )))))
37 (reset! open-or-prepare-was-called? true)
38 (log-message "Prepared bolt " component-id ":" (keys task-datas))
39 (setup-metrics! executor-data)
40
41 (let [receive-queue (:receive-queue executor-data)
42     event-handler (mk-task-receiver executor-data tuple-action-fn)]
43     (disruptor/consumer-started! receive-queue)))
```

- 第7~10行主要获取Bolt对象并定义相关方法。bolt-emit方法用于向Executor的消息发送队列中发送消息，是其中最重要的方法，前面简要介绍过。
- 第12~36行调用Bolt对象的prepare方法，同时实例化Bolt对象的OutputCollector对象作为prepare方法的传入参数，OutputCollector的emit方法将调用bolt-emit函数来发送消息，ack及fail方法则用来对消息进行跟踪。这里，ack和fail方法中采用了task.clj的send-unanchored方法向Acker Bolt的相关流发送消息，详情请参考第12章。
- 第42行调用mk-task-receiver函数来获得接收队列的处理函数。

10.4.5 消息循环

主消息循环比较简单，它调用阻塞方式的consume-batch-when-available函数对接收队列中

的消息进行处理，相关代码如下：

```
(fn []
    (disruptor/consume-batch-when-available receive-queue event-handler)
    0)
```

10.5 创建 Executor

最后来看Worker是如何创建一个Executor的，其中涉及的mk-executor函数的代码如下：

```
1 (defn mk-executor [worker executor-id]
2    (let [executor-data (mk-executor-data worker executor-id)
3            _ (log-message "Loading executor " (:component-id executor-data) ":" (pr-str
                executor-id))
4            task-datas (->> executor-data
5                            :task-ids
6                            (map (fn [t] [t (task/mk-task executor-data t)]))
7                            (into {})
8                            (HashMap.))
9            _ (log-message "Loaded executor tasks " (:component-id executor-data) ":" (pr-str
                executor-id))
10           report-error-and-die (:report-error-and-die executor-data)
11           component-id (:component-id executor-data)
12
13           ;; starting the batch-transfer->worker ensures that anything publishing to that
                queue
14           ;; doesn't block (because it's a single threaded queue and the caching/consumer
                started
15           ;; trick isn't thread-safe)
16           system-threads [(start-batch-transfer->worker-handler! worker executor-data)]
17           handlers (with-error-reaction report-error-and-die
18                         (mk-threads executor-data task-datas))
19           threads (concat handlers system-threads)]
20     (setup-ticks! worker executor-data)
21
22     (log-message "Finished loading executor " component-id ":" (pr-str executor-id))
23     ;; TODO: add method here to get rendered stats... have worker call that when heartbeating
24     (reify
25         RunningExecutor
26         (render-stats [this]
27            (stats/render-stats! (:stats executor-data)))
28         (get-executor-id [this]
29            executor-id )
30         Shutdownable
31         (shutdown
32            [this]
33            (log-message "Shutting down executor " component-id ":" (pr-str executor-id))
34            (disruptor/halt-with-interrupt! (:receive-queue executor-data))
35            (disruptor/halt-with-interrupt! (:batch-transfer-queue executor-data))
36            (doseq [t threads]
37                (.interrupt t)
```

```
38                (.join t))
39
40            (doseq [user-context (map :user-context (vals task-datas))]
41                (doseq [hook (.getHooks user-context)]
42                    (.cleanup hook)))
43            (.disconnect (:storm-cluster-state executor-data))
44            (when @(:open-or-prepare-was-called? executor-data)
45                (doseq [obj (map :object (vals task-datas))]
46                    (close-component executor-data obj)))
47            (log-message "Shut down executor " component-id ":" (pr-str executor-id)))
48    )))
```

- 第2行调用mk-executor-data来创建Executor的数据。
- 第4~8行则调用mk-task来创建Executor中每个Task对应的数据。
- 第16行调用start-batch-transfer->worker-handler!方法启动Executor的数据发送线程。
- 第17~18行调用mk-threads方法来获得Executor的主循环线程，并通过with-error-reaction宏对mk-threads进行包装。当异常发生时，调用report-error-and-die方法记录错误并退出。
- 第25~29行实例化RunningExecutor对象用以操作Executor，例如调用stats/render-stats!函数来收集Executor的运行统计等。
- 第31~48行实例化Shutdownable，用于退出Executor并清理相关资源，具体的操作包括：
  - 结束Disruptor Queue的消息循环；
  - 结束Executor中启动的线程；
  - 清理用户钩子的数据；
  - 断开到ZooKeeper的连接；
  - 依次调用Executor中Spout或Bolt的close方法。

这些操作并不一定会被全部调用。

10.6　辅助函数介绍

在这一节中，我们介绍一下Executor中用到的一些重要辅助方法。

10.6.1　组件的Grouper函数

获得当前组件中一个流的所有接收端及其接收方式是Executor中的重要算法，这是tasks-fn函数完成各种操作的前提。

outbound-components函数用于获取从组件到某一个流的分组函数，tasks-fn函数通过调用该分组函数便可获得消息的目标Task集合。outbound-components的代码如下：

```
1 (defn outbound-components
2   "Returns map of stream id to component id to grouper"
3   [^WorkerTopologyContext worker-context component-id]
4   (->> (.getTargets worker-context component-id)
5        clojurify-structure
```

```
6        (map (fn [[stream-id component->grouping]]
7               [stream-id
8                 (outbound-groupings
9                      worker-context
10                     component-id
11                     stream-id
12                     (.getComponentOutputFields worker-context component-id stream-id)
13                     component->grouping)]))
14       (into {})
15       (HashMap.)))
```

- 第4行调用`WorkerTopologyContext`对象的`getTargets`方法得到一个哈希表，该哈希表的键为当前组件所对应的流，值也为一个哈希表，用于记录目标组件以何种方式从该流接收数据。
- 第8行调用`outbound-groupings`函数通过Thrift类型的分组方式来获得分组函数。

`outbond-groupings`函数的定义如下：

```
1 (defn- outbound-groupings [^WorkerTopologyContext worker-context this-component-id stream-id
    out-fields component->grouping]
2   (->> component->grouping
3       (filter-key #(-> worker-context
4                        (.getComponentTasks %)
5                        count
6                        pos?))
7       (map (fn [[component tgrouping]]
8               [component
9                 (mk-grouper worker-context
10                       this-component-id
11                       stream-id
12                       out-fields
13                       tgrouping
14                       (.getComponentTasks worker-context component)
15                       )]))
16       (into {})
17       (HashMap.)))
```

- 第3~6行对目标组件进行过滤，若组件对应的TaskId集合为空，则会被过滤掉。`filter-key`函数的功能是对一个哈希表中的键进行过滤。
- 第7~15行利用`map`函数对组件及其分组方式进行处理，调用`mk-grouper`函数来产生分组函数，并最终返回一个保存有从组件到分组函数的映射关系的哈希表。

`mk-grouper`函数是Executor的核心方法。它会返回一个函数，该函数返回一个TaskId集合，代表消息发送的目的Task集合。`mk-grouper`的代码如下：

```
1 (defn- mk-grouper
2    "Returns a function that returns a vector of which task indices to send tuple to, or just a
        single task index."
3    [^WorkerTopologyContext context component-id stream-id ^Fields out-fields thrift-grouping
        ^List target-tasks]
4    (let [num-tasks (count target-tasks)
```

```
5              random (Random.)
6              target-tasks (vec (sort target-tasks))]
7         (condp = (thrift/grouping-type thrift-grouping)
8            :fields
9              (if (thrift/global-grouping? thrift-grouping)
10                (fn [task-id tuple]
11                   ;; It's possible for target to have multiple tasks if it reads multiple
                       sources
12                   (first target-tasks))
13                (let [group-fields (Fields. (thrift/field-grouping thrift-grouping))]
14                   (mk-fields-grouper out-fields group-fields target-tasks)
15                   ))
16           :all
17               (fn [task-id tuple] target-tasks)
18           :shuffle
19              (mk-shuffle-grouper target-tasks)
20           :local-or-shuffle
21              (let [same-tasks (set/intersection
22                       (set target-tasks)
23                       (set (.getThisWorkerTasks context)))]
24                 (if-not (empty? same-tasks)
25                     (mk-shuffle-grouper (vec same-tasks))
26                     (mk-shuffle-grouper target-tasks)))
27           :none
28              (fn [task-id tuple]
29                 (let [i (mod (.nextInt random) num-tasks)]
30                    (.get target-tasks i)
31                    ))
32           :custom-object
33               (let [grouping (thrift/instantiate-java-object (.get_custom_object
                   thrift-grouping))]
34                   (mk-custom-grouper grouping context component-id stream-id target-tasks))
35           :custom-serialized
36               (let [grouping (Utils/deserialize (.get_custom_serialized thrift-grouping))]
37                   (mk-custom-grouper grouping context component-id stream-id target-tasks))
38           :direct
39               :direct
40           )))
```

- 第4行和第6行分别获得与目标组件对应的Task的数目以及排列后的列表，它们将作为计算目标Task的函数输入。不过某些分组方式只需要目标组件的Task数目即可，例如ShuffleGrouping操作。
- 第7行针对Thrift类型的不同分组方式分别构建分组函数。

下面我们来详细介绍Storm提供的几种分组方式。

1. 字段分组

该分组方式首先通过消息中与分组字段名列表对应的值计算得到一个哈希值，然后将该哈希值模除目标节点的个数，选出这条消息的目标节点。在这种分组方式下，每条消息只会到达某一个目标节点，相关代码如下：

```
1 (defn- mk-fields-grouper [^Fields out-fields ^Fields group-fields ^List target-tasks]
2     (let [num-tasks (count target-tasks)
3             task-getter (fn [i] (.get target-tasks i))]
4         (fn [task-id ^List values]
5             (-> (.select out-fields group-fields values)
6                 tuple/list-hash-code
7                 (mod num-tasks)
8                 task-getter))))
```

第3行定义一个函数task-getter，它根据下标从target-tasks中取出实际的TaskId。第4~8行定义了实际的分组函数，它实际上调用消息（Java List对象）的hashCode方法得到一个数值并抹除接收端Task的数目，从而获得目标Task的索引。list-hash-code函数的代码如下：

```
(defn list-hash-code [^List alist]
    (.hashCode alist))
```

这里要注意的是：如果List中的元素为数组，此时的哈希值实际上是按照数组地址而非具体数据来进行计算的，这可能会导致内容相同的消息无法到达相同的节点。这种情况下，用户应使用基于数据而非地址的哈希方法。

如果传入的分组字段列表为空，第6行代码得到的哈希值将为1，于是所有的发送消息都会被送到一个节点上去，该分组方式就演化成了全局分组（Global Grouping）。不过哈希值1实际上表示目标节点中的第2个节点，这可能会导致误解或者错误，因此最好不要传入空的分组字段列表。

2. 随机分组

该分组方式表示将输入消息随机地分配到目标节点上，相关代码如下：

```
1  (defn- mk-shuffle-grouper [^List target-tasks]
2      (let [choices (rotating-random-range target-tasks)]
3          (fn [task-id tuple]
4             (acquire-random-range-id choices))))
5
6  (defn rotating-random-range [choices]
7      (let [rand (Random.)
8               choices (ArrayList. choices)]
9          (Collections/shuffle choices rand)
10         [(MutableInt. -1) choices rand]))
11
12 (defn acquire-random-range-id [[^MutableInt curr ^List state ^Random rand]]
13     (when (>= (.increment curr) (.size state))
14         (.set curr 0)
15         (Collections/shuffle state rand))
16     (.get state (.get curr)))
```

- 第1~4行定义随机分组函数，该函数只需传入目标的节点列表。该函数会调用rotating-random-range进行初始化，即通过在函数体中不断调用函数acquire-random-range-id返回所有目标节点的TaskId。这个算法保证了每一个节点都有均等的机会收到新消息，同时它也具有较好的随机性。

- 第6~10行定义了一个通用函数rotating-random-range，该函数会调用输入集合的搅乱操作，返回值为{–1,搅乱过后的集合，一个随机数}。
- 第12~16行定义了一个通用函数acquire-random-range-id，其输入为上一个函数的输出，输出为变量curr指示的元素。该函数会自增curr，并判断它是否大于目标节点集合的大小，若超出了范围就将其重置为0，然后重新搅乱输入的目标节点集合，这样增加了更多的随机性。

3. 全部分组

该分组方式表示，所有的接收端节点都会收到待发消息，相关代码如下：

```
(fn [task-id tuple] target-tasks)
```

4. 无分组

在该分组方式下，Storm将随机选择一个目标节点，并将所有的消息都将发送到该节点上，相关代码如下：

```
(fn [task-id tuple]
    (let [i (mod (.nextInt random) num-tasks)]
        (.get target-tasks i)
        ))
```

在Topology启动后，上面的代码将通过利用一个随机数模除目标Task数的手段来选择接收节点，之后要发送的消息都将发送至该节点。

5. 直接分组

在这种分组方式下，消息将直接发送到某一个特定节点，这在传递控制信息时非常有用。在事务Topology中经常可以看到这种分组方式。

6. 自定义分组

这是用户自定义的分组方法。用户需要实现下面的CustomStreamGrouping接口来完成自定义：

```
public interface CustomStreamGrouping extends Serializable {

   /**
     * Tells the stream grouping at runtime the tasks in the target bolt.
     * This information should be used in chooseTasks to determine the target tasks.
     *
     * It also tells the grouping the metadata on the stream this grouping will be used on.
     */
   void prepare(WorkerTopologyContext context, GlobalStreamId stream, List<Integer> targetTasks);

   /**
     * This function implements a custom stream grouping. It takes in as input
     * the number of tasks in the target bolt in prepare and returns the
     * tasks to send the tuples to.
     *
     * @param values the values to group on
     */
```

```
    List<Integer> chooseTasks(int taskId, List<Object> values);
}
```

这里向prepare函数中传入目标节点集合以及当前组件的元信息，是为了给消息接收端的计算提供上下文。

chooseTasks函数将在运行时调用，其参数为当前的TaskId和消息内容。

那么用户自定义的分组函数是如何集成到storm中的呢？mk-custom-grouper函数将定义一个分组函数，该分组函数的实现是基于用户自定义的实现了CustomStreamGrouping接口的类，其代码如下：

```
1 (defn- mk-custom-grouper [^CustomStreamGrouping grouping ^WorkerTopologyContext context ^String
     component-id ^String stream-id target-tasks]
2    (.prepare grouping context (GlobalStreamId. component-id stream-id) target-tasks)
3    (fn [task-id ^List values]
4       (.chooseTasks grouping task-id values)
5       ))
```

- 第2行调用用户实现的prepare方法，传入当前的GlobalStreamId及目标Task列表，以准备分组函数的上下文。
- 第3~5行定义并返回一个函数，该函数将调用用户实现的chooseTasks来选择一个目标节点。

7. Local_or_Shuffle Grouping

这种分组方式会首先选择将消息发送到属于同一Worker的目标Task上。由于属于同一进程空间，被发送的消息将不需要经过ZMQ及网络来传输，而是通过Disruptor Queue在线程之间传输，这样性能会得到提高，相关代码如下：

```
(let [same-tasks (set/intersection
         (set target-tasks)
         (set (.getThisWorkerTasks context)))]
    (if-not (empty? same-tasks)
        (mk-shuffle-grouper (vec same-tasks))
        (mk-shuffle-grouper target-tasks)))
```

如果没有找到这样的Task，就通过随机分组方式来选择消息的接收端。

10.6.2 带流量控制的错误报告方法

Storm会将Executor产生的错误记录到ZooKeeper中，而Nimbus通过访问ZooKeeper获得该数据并将其显示在StormUI上。过于频繁以及过多的错误报告都会对ZooKeeper造成影响，因此Storm采用了相应的控制策略，即限制同一时间段内报告错误的数量，并对ZooKeeper中老数据进行清理，相关代码如下：

```
1 (defn throttled-report-error-fn [executor]
2    (let [storm-conf (:storm-conf executor)
3            error-interval-secs (storm-conf TOPOLOGY-ERROR-THROTTLE-INTERVAL-SECS)
4            max-per-interval (storm-conf TOPOLOGY-MAX-ERROR-REPORT-PER-INTERVAL)
5            interval-start-time (atom (current-time-secs))
6            interval-errors (atom 0)
```

```
7             ]
8         (fn [error]
9         (log-error error)
10        (when (> (time-delta @interval-start-time)
11                   error-interval-secs)
12          (reset! interval-errors 0)
13          (reset! interval-start-time (current-time-secs)))
14        (swap! interval-errors inc)
15
16        (when (<= @interval-errors max-per-interval)
17        (cluster/report-error (:storm-cluster-state executor) (:storm-id executor) (:component-id
            executor) error)
18            ))))
```

- 第3行中error-interal-secs的值是从Storm的配置项获得的，默认为10秒钟。第4行的max-per-interval也是从Storm的配置项获得的，默认值为5。这两个参数的含义为10秒钟最多可以报告错误5次。
- 第8~18行定义错误报告函数。第9行首先对错误进行日志记录。若当前时间已经超出了error-interval-secs，则重新开始一个新的统计区间，并将interval-errors重置为0。第17行调用cluster/report-error将错误记录到ZooKeeper中。其中，report-error方法的定义如下：
- 第2行获得ZooKeeper中存储错误的路径path，其默认值为/storm/errors/<storm-id>/<component-id>。可以看出，错误是按照组件来分别存储的。
- 第3行构建数据，:time-secs为当前的时间，:error为异常的错误栈。
- 第4~5行创建父节点以及用于存储本次错误的ZooKeeper节点，其中节点为Sequential类型。
- 第6~9行获得该组件对应的错误列表，然后移除最近的10条错误信息，并将剩余的错误信息保存到to-kill变量中。第10~11行将to-kill数据从ZooKeeper中清理掉。于是，ZooKeeper始终保存了每一个组件最近的10条错误消息。实际上，我认为这里应该再设置一个时间窗口，将更为久远的错误消息也清除掉。

```
1 (report-error [this storm-id component-id error]
2    (let [path (error-path storm-id component-id)
3            data {:time-secs (current-time-secs) :error (stringify-error error)}
4            _ (mkdirs cluster-state path)
5            _ (create-sequential cluster-state (str path "/e") (Utils/serialize data))
6            to-kill (->> (get-children cluster-state path false)
7                          (sort-by parse-error-path)
8                          reverse
9                          (drop 10))]
10       (doseq [k to-kill]
11           (delete-node cluster-state (str path "/" k)))))
```

10.6.3　触发系统Ticks

setup-ticks！函数用来定期向该Execturor的接收消息队列发送Tick消息。Executor在收到T：ck消息之后，就会执行发送队列的超时操作。因此，setup-ticks！主要用于对Spout节点发送出

去的消息进行超时操作，相关代码如下：

```
1 (defn setup-ticks! [worker executor-data]
2     (let [storm-conf (:storm-conf executor-data)
3             tick-time-secs (storm-conf TOPOLOGY-TICK-TUPLE-FREQ-SECS)
4             receive-queue (:receive-queue executor-data)
5             context (:worker-context executor-data)]
6         (when tick-time-secs
7             (if (and (not (storm-conf TOPOLOGY-ENABLE-MESSAGE-TIMEOUTS))
8                 (= :spout (:type executor-data)))
9                 (log-message "Timeouts disabled for executor " (:executor-id executor-data))
10                (schedule-recurring
11                   (:user-timer worker)
12                   tick-time-secs
13                   tick-time-secs
14                   (fn []
15                      (disruptor/publish
16                          receive-queue
17                          [[nil (TupleImpl. context [tick-time-secs] -1
                               Constants/SYSTEM_TICK_STREAM_ID)]]
18                          )))))))
```

- 在第3~5行中，配置项`TOPOLOGY-TICK-TUPLE-FREQ-SECS`用来控制向`__system`流以及`__tick`流发送消息的频率，`tick-time-secs`用来保存该频率值。`receive-queue`为Executor对应的接收Disruptor Queue，`context`为`WorkerTopologyContext`对象。可以看出，Tick消息只发送到本地Worker，并不能被其他Worker的Executor收到。
- 在第6行中，若设置`tick-time-secs`，则开始设置系统的Tick消息。
- 在第7~9行中，若该节点为Spout节点并且未设置消息超时，则打印消息并退出。参数`TOPOLOGY-ENABLE-MESSAGE-TIMEOUTS`主要用于调试模式，由于超时的消息会给系统调试带来额外的复杂性，因此可在调试过程中暂时关闭消息的超时操作。当Spout收到Tick消息时，可以对缓存在`pending`对象中的数据进行超时操作。
- 第10~18行利用Worker定义的用户计时器以`tick-time-secs`为间隔设置计时器。第14~18行定义计时器的回调函数，并向`receive-queue`中发送一条消息。从第17行可以看出，该消息对应的TaskId为`nil`，表示该Executor中所有的Task都会收到该消息；消息的内容为`tick-time-secs`；-1表示系统TaskId；最后一项表示该消息会被发送到`SYSTEM_TICK_STREAMID`。

Executor在创建DisruptorQueue的处理器时，若消息的TaskId为`nil`，Executor将发送消息到其所有Task的`tuple-action-fn`，相关代码如下：

```
1 (defn mk-task-receiver [executor-data tuple-action-fn]
2     (let [^KryoTupleDeserializer deserializer (:deserializer executor-data)
3             task-ids (:task-ids executor-data)
4             debug? (= true (-> executor-data :storm-conf (get TOPOLOGY-DEBUG)))
5             ]
6         (disruptor/clojure-handler
7             (fn [tuple-batch sequence-id end-of-batch?]
8                (fast-list-iter [[task-id msg] tuple-batch]
```

```
9                   (let [^TupleImpl tuple (if (instance? Tuple msg) msg (.deserialize
                        deserializer msg))]
10                      (when debug? (log-message "Processing received message " tuple))
11                      (if task-id
12                         (tuple-action-fn task-id tuple)
13                         ;; null task ids are broadcast tuples
14                         (fast-list-iter [task-id task-ids]
15                             (tuple-action-fn task-id tuple)
16                             ))
17                      ))))))
```

- 第2~5行获得该Executor对应的TaskId列表以及消息的反序列化器，默认为KryoTupleDeserializer类型。若设置TOPOLOGY-DEBUG为true，系统将打印每一条其收到和发送的消息。
- 第6~17行实现DisruptorQueue的消息处理回调函数。tuple-batch中含有收到的一组消息，其消息格式为[task-id, msg]。第9行调用反序列化器将msg反序列化为TupleImpl对象。第11行根据是否已设置task-id来决定该消息是发送给某一个TaskId还是发送到该Executor中所有TaskId。
- 发送到SYSTEM-TICK-STREAM-ID流的消息为Tick消息，Task收到Tick消息后，将调用metrics-tick函数来处理。

10.7　小结

Executor中的通信关系如图10-1所示。

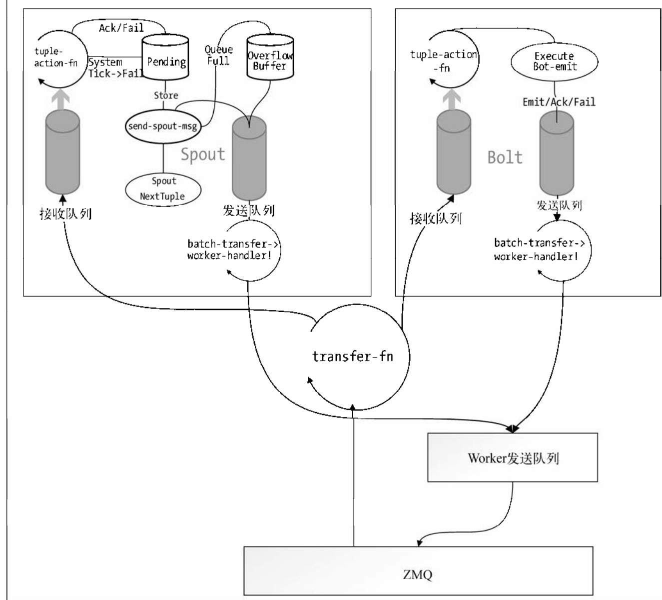


图10-1　Executor的架构示意图

下面简要介绍一下：

- Worker从ZMQ收到消息后，会按照TaskId将消息分发到Executor的消息接收队列中。
- Executor的主消息循环负责处理消息并将产生的消息发送到Executor的消息发送队列中。
- 每个Executor都有一个发送线程用以监听发送队列，它会将发送队列中的消息发送到Worker的消息发送队列或者其他Executor的接收队列中。Worker中的所有Executor会共享Worker的消息发送队列。

第 11 章 Task

Storm中的Task是最小的执行单位。与Worker、Executor分别对应于进程和线程不同，Task只是逻辑上的执行单位，它需要寄身于Executor中完成运行。一个Executor可以含有多个Task。用户定义的Spout和Bolt对象都会被放置在Task上。当Executor收到属于某一个Task的消息时，就会调用与该Task对应的Spout或Bolt对象的相关方法进行处理。

11.1 Task 的上下文对象

TopologyContext对象为Task的执行提供了上下文环境，它实际上给出了诸如Topology的结构、流的定义等与Topology相关的信息。

11.1.1 TopologyContext

TopologyContext类继承自WorkerTopologyContext，它对应于某一个Task运行的上下文Bolt的prepare方法以及Spout的open方法均会传入该类的对象。该类的代码如下：

```
public class TopologyContext extends WorkerTopologyContext implements IMetricsContext {
    private Integer _taskId;
    private Map<String, Object> _taskData = new HashMap<String, Object>();
    private List<ITaskHook> _hooks = new ArrayList<ITaskHook>();
    private Map<String, Object> _executorData;
    private Map<Integer,Map<Integer, Map<String, IMetric>>> _registeredMetrics;
private clojure.lang.Atom _openOrPrepareWasCalled;
}
```

下面简要介绍该类中的成员变量。

- _taskId为该上下文对象对应的TaskId。
- _taskData为该Task共享的数据。
- _executorData为Task所在Executor共享的数据，用于在属于同一Executor的Task之间共享数据。
- _hooks是Storm为用户提供的一种扩展机制。用户可以为Bolt或者Spout对象添加相应的回调钩子，当特定的事件发生时，钩子方法将被调用。TopologyContext对象采用addTaskHooks来添加一个钩子，利用getHooks获得钩子回调函数。这样，用户便可更加灵活地进行一些

运行时统计工作了。Executor会在适当的时机调用这些钩子方法。这里涉及的ITaskHook接口的定义如下：

```
public interface ITaskHook {
    void prepare(Map conf, TopologyContext context);
    void cleanup();
    void emit(EmitInfo info);
    void spoutAck(SpoutAckInfo info);
    void spoutFail(SpoutFailInfo info);
    void boltExecute(BoltExecuteInfo info);
    void boltAck(BoltAckInfo info);
    void boltFail(BoltFailInfo info);
}
```

例如对于某一个Bolt节点，用户可以实现boltExecute方法，该方法传入的参数为BoltExecuteInfo对象，其中含有执行的消息、taskId以及执行时间等信息，用户可以根据这些信息完成一些自定义功能。目前，这个功能的用处还不多。BoltExecuteInfo类的定义如下：

```
public class BoltExecuteInfo {
    public Tuple tuple;
    public int executingTaskId;
    public Long executeLatencyMs; // null if it wasn't sampled
}
```

回到前面TopologyContext的定义中，这里的_registeredMetrics和_openOrPrepareWasCalled主要用于系统的内置统计信息中。例如，在SystemBolt中注册的许多与Task所在Worker相关的信息就属于这类内置统计信息。

接下来我们讨论TopologyContext的父类中所含有的数据。

11.1.2 GeneralTopologyContext

GeneralTopologyContext类表示Topology的上下文环境，它提供较多的工具方法来方便获得Topology的结构信息。该类的定义如下：

```
public class GeneralTopologyContext implements JSONAware {
    private StormTopology _topology;
    private Map<Integer, String> _taskToComponent;
    private Map<String, List<Integer>> _componentToTasks;
    private Map<String, Map<String, Fields>> _componentToStreamToFields;
    private String _stormId;
    protected Map _stormConf;
}
```

下面简要介绍各个成员变量的含义。

- _topology为Thrift生成的Storm Topology类型的对象，其中含有Bolt和Spout的输入输出等信息。
- _taskToComponent为从TaskId到组件ID的映射。

- _componentToTasks为从组件ID到其对应的Task集合的映射。
- _componentToStreamToFields为从组件到每个输出流的模式的映射。
- _stormId和_stormConf分别表示Topology的id和配置项。

GeneralTopologyContext中定义的主要工具函数如表11-1所示。

表11-1　GeneralTopologyContext的工具函数

工具函数	描　　述
String getComponentId(int taskId)	根据taskId获得其所在组件的名字
Set<String> getComponentStreams(String componentId)	根据组件名字获得该组件对应的输出流
List<Integer> getComponentTasks(String componentId)	根据组件名字获得其TaskId的集合
Fields getComponentOutputFields(String componentId, String streamId)	根据组件ID以及流序号获得该流的模式
Fields getComponentOutputFields(Global StreamId id)	返回值同上。
	利用组件ID以及流ID可以构建GlobalStreamId对象
Map<GlobalStreamId, Grouping> getSources (String componentId)	获得组件的输入。根据组件序号获得该组件输入的GlobalStreamId以及分组方式
Map<String, Map<String, Grouping>> get Targets(String componentId)	获取输出节点，键为当前组件的输出流序号，值为目标节点的组件ID以及分组方式
Map<Integer, String> getTaskTo Component()	返回从Task到组件的映射关系
Set<String> getComponentIds()	获得所有的组件ID集合
int maxTopologyMessageTimeout()	获取Topology的最大超时时间，由TOPOLOGY_MESSAGE_TIMEOUT_SECS参数得到，根据Topology的配置文件以及各个Spout节点单独设置的最大值计算获得

在上述工具函数，getSources和getTargets分别用来获取一个组件的输入及输出，是其中尤为重要的方法。

11.1.3　WorkerTopologyContext

WorkerTopologyContext类继承自GeneralTopologyContext。顾名思义，它是Storm中Worker运行的上下文环境，也即Executor中的共享环境。该类的实现代码如下：

```
public class WorkerTopologyContext extends GeneralTopologyContext {
    public static final String SHARED_EXECUTOR = "executor";
    private Integer _workerPort;
    private List<Integer> _workerTasks;
    private String _codeDir;
    private String _pidDir;
    Map<String, Object> _userResources;
    Map<String, Object> _defaultResources;
}
```

下面简要介绍该类中的成员变量。

- _workerPort为该Worker所对应的端口号，用于ZMQ传输数据等。
- _workerTasks为该Worker上执行的Task集合。
- _codeDir为第三方代码的目录，是启动其他语言程序的工作目录，具体内容为$StormData\supervisor\stormdist\[TopologyId]\resources。
- _pidDir为该Worker进程所对应的目录，所有由该Worker产生的子进程都会在该目录下写下进程编号，待到Worker结束时就将杀掉对应的子进程。这个目录为$StormData\workers\[uuid]\pids\，其中$StormData由配置项决定，而[uuid]为一个随机的GUID值，代表了Worker的ID。
- _userResources为Executor的共享资源，目前没有用到。
- _defaultResources为默认资源，目前含有一个线程池。mk-default-resources函数为Worker创建了一个大小为TOPOLOGY-WORKER-SHARED-THREAD-POOL-SIZE的线程池，并放置于_defaultResources中，其键为executor。mk-default-resources的代码如下：

```
(defn- mk-default-resources [worker]
    (let [conf (:conf worker)
        thread-pool-size (int (conf TOPOLOGY-WORKER-SHARED-THREAD-POOL-SIZE))]
        {WorkerTopologyContext/SHARED_EXECUTOR (Executors/newFixedThreadPool thread-pool-size)}
        ))
```

用户可以调用getSharedExecutor来获得该线程池并使用它。

```
public ExecutorService getSharedExecutor() {
    return (ExecutorService) _defaultResources.get(SHARED_EXECUTOR);
}
```

11.1.4 TopologyContext

在创建一个Task时，需要为其先创建一个TopologyContext对象，以便可以更加容易地获得Topology信息，例如系统中的组件、组件对应的Task等。mk-topology-context-builder函数用来创建该对象，其代码如下：

```
1 (defn mk-topology-context-builder [worker executor-data topology]
2     (let [conf (:conf worker)]
3         #(TopologyContext.
4             topology
5             (:storm-conf worker)
6             (:task->component worker)
7             (:component->sorted-tasks worker)
8             (:component->stream->fields worker)
9             (:storm-id worker)
10            (supervisor-storm-resources-path
11               (supervisor-stormdist-root conf (:storm-id worker)))
12            (worker-pids-root conf (:worker-id worker))
13            (int %)
14            (:port worker)
15            (:task-ids worker)
```

```
16          (:default-shared-resources worker)
17          (:user-shared-resources worker)
18          (:shared-executor-data executor-data)
19          (:interval->task->metric-registry executor-data)
20          (:open-or-prepare-was-called? executor-data))))
```

该函数较为直观，其中第13行将传入的参数转换成为int类型，%表示占位符，传入的参数为该Task的TaskId。

11.2　创建 Task 数据

mk-task-data函数用来创建与Task相关的数据，其参数为Executor的数据executor-data以及该Task的TaskId，其代码如下：

```
1 (defn mk-task-data [executor-data task-id]
2    (recursive-map
3        :executor-data executor-data
4        :task-id task-id
5        :system-context (system-topology-context (:worker executor-data) executor-data task-id)
6        :user-context (user-topology-context (:worker executor-data) executor-data task-id)
7        :builtin-metrics (builtin-metrics/make-data (:type executor-data))
8        :tasks-fn (mk-tasks-fn <>)
9           object (get-task-object (.getRawTopology ^TopologyContext (:system-context <>))
               (:component-id executor-data))))
```

- :executor-data为Task所在的Executor的数据。
- :task-id为该Task的序号，在整个Topology中是唯一的。
- :system-context通过调用system-topology-context方法获得。该方法会调用mk-topology-context-builder函数创建TopologyContext对象，其传入的Topology对象是Worker通过system-topology函数产生的。system-topology函数的功能为在用户定义的Topology的基础上添加系统组件（如Acker Bolt等）。
- :user-context的产生方法与:system-context类似，只不过它是通过传入Worker的:topology对象来构建TopologyContext的，因此只包含用户定义的Topology。
- :builtin-metrics使用builtin-metrics/make-data并根据Executor的类型来创建内置的统计信息如消息发送数目等。
- :tasks-fn是Task的核心数据成员，它通过mk-tasks-fn函数产生，主要用来选择消息的目标节点以及发送消息。
- :object对应于该Task所执行的Spout或Bolt对象。这里首先调用system-context的getRawTopology获取用户定义的Topology，即user-context中的Topology对象，之后调用get-task-object方法，根据组件ID获取该组件所对应的Spout或Bolt对象。

  get-task-object函数用于获取Task所对应的Spout或Bolt对象，其代码如下：

```
1 (defn- get-task-object [^TopologyContext topology component-id]
2    (let [spouts (.get_spouts topology)
3             bolts (.get_bolts topology)
4             state-spouts (.get_state_spouts topology)
5             obj (Utils/getSetComponentObject
6                     (cond
7                        (contains? spouts component-id) (.get_spout_object ^SpoutSpec (get spouts
                           component-id))
8                        (contains? bolts component-id) (.get_bolt_object ^Bolt (get
                           bolts component-id))
9                  (contains? state-spouts component-id) (.get_state_spout_object
                     ^StateSpoutSpec
                     (get state-spouts component-id))
10                       true (throw-runtime "Could not find " component-id " in " topology)))
11            obj (if (instance? ShellComponent obj)
12                (if (contains? spouts component-id)
13                    (ShellSpout. obj)
14                    (ShellBolt. obj))
15                obj )
16            obj (if (instance? JavaObject obj)
17                (thrift/instantiate-java-object obj)
18                obj )]
19     obj
20     ))
21 public static Object getSetComponentObject(ComponentObject obj) {
22     if(obj.getSetField()==ComponentObject._Fields.SERIALIZED_JAVA) {
23         return Utils.deserialize(obj.get_serialized_java());
24     } else if(obj.getSetField()==ComponentObject._Fields.JAVA_OBJECT) {
25         return obj.get_java_object();
26     } else {
27         return obj.get_shell();
28     }
29 }
```

- 第2~4行获得所有用户定义的spouts、bolts和state-spouts对象，每个组件只会属于其中一个集合。
- 第5~10行获取对应的ComponentObject类型，并调用getSetComponentObject方法将其转化为实际对象。
- 第11~15行进行判断，若obj为ShellComponent，则调用ShellSpout或者ShellBolt进行封装，传入的obj实际上为ShellSpout或者ShellBolt要启动的命令。在Storm中，我们利用ShellSpout以及ShellBolt来完成多语言处理，它们内部含有需要执行的命令。
- 第16~18行首先进行判断，若obj为JavaObject对象，并且该对象含有full_class_name以及args_list成员变量，则利用instantiate-java-object函数来创建JavaObject所代表的目标对象。instantiate-java-object函数的定义如下：

```
(defn instantiate-java-object [^JavaObject obj]
    (let [name (symbol (.get_full_class_name obj))
             args (map (memfn getFieldValue) (.get_args_list obj))]
        (eval `(new ~name ~@args))
        ))
```

11

11.3　mk-tasks-fn 函数

mk-tasks-fn函数将返回一个函数tasks-fn，tasks-fn函数可根据要发送的消息获取消息的接收端TaskId集合。为了便于讨论，这里将用于运行统计的代码去掉，详情可参见第13章和第14章。mk-tasks-fn函数的代码如下：

```
1 (defn mk-tasks-fn [task-data]
2    (let [task-id (:task-id task-data)
3         executor-data (:executor-data task-data)
4         component-id (:component-id executor-data)
5         ^WorkerTopologyContext worker-context (:worker-context executor-data)
6         storm-conf (:storm-conf executor-data)
7         emit-sampler (mk-stats-sampler storm-conf)
8         stream->component->grouper (:stream->component->grouper executor-data)
9         user-context (:user-context task-data)
10        executor-stats (:stats executor-data)
11        debug? (= true (storm-conf TOPOLOGY-DEBUG))]
12
13       (fn ([^Integer out-task-id ^String stream ^List values]
14          (when debug?
15             (log-message "Emitting direct: " out-task-id "; " component-id " " stream " "
                   values))
16          (let [target-component (.getComponentId worker-context out-task-id)
17                component->grouping (get stream->component->grouper stream)
18                grouping (get component->grouping target-component)
19                out-task-id (if grouping out-task-id)]
20               (when (and (not-nil? grouping) (not= :direct grouping))
21               (throw (IllegalArgumentException. "Cannot emitDirect to a task expecting a
                   regulargrouping")))
22               (if out-task-id [out-task-id])
23               ))
24       ([^String stream ^List values]
25          (when debug?
26             (log-message "Emitting: " component-id " " stream " " values))
27          (let [out-tasks (ArrayList.)]
28             (fast-map-iter [[out-component grouper] (get stream->component->grouper stream)]
29                (when (= :direct grouper)
30                    ;;  TODO: this is wrong, need to check how the stream was declared
31                   (throw (IllegalArgumentException. "Cannot do regular emit to direct
                       stream")))
32                (let [comp-tasks (grouper task-id values)]
33                   (if (or (sequential? comp-tasks) (instance? Collection comp-tasks))
34                   (.addAll out-tasks comp-tasks)
35                   (.add out-tasks comp-tasks)
36                   )))
37               out-tasks)))
38    ))
```

- 第8行定义的stream->component->grouper变量十分重要，它指定了系统中的组件将以分组函数定义的方式去收听流。
- 第13~23行定义函数的第一个重载。该重载函数用于应对向直接流发送消息的情况，传入的参数中out-task-id为接收端的TaskId，stream为消息发送流，values为要发送的消息。

当用户将TOPOLOGY_DEBUG设置为true时，debug?方法也将返回true，此时系统将对输入的消息进行记录。

- 第16~19行通过out-task-id获得其所在的组件以及该组件对该直接流的收听方式。直接流只能采取直接分组的方式进行收听，第20~23行对此进行检测。注意若out-task-id为空，则表示该流没有组件收听，这在Storm中是允许的，该消息将被丢弃掉。由于发送到直接流的消息的目标Task已经确定，因此这部分重载的最主要功能其实是完成异常检测。
- 第24~38行定义函数的另外一个重载，其输入参数仅包括消息所在的流以及消息本身。第28行根据流找到所有收听该流的组件及其收听方式。类似地，对于非直接流，不能使用直接分组的方式进行收听。
- 第32行调用grouper函数来获得目标节点的TaskId，输入为发送消息的Task对应的TaskId以及消息本身。例如，随机分组的分组函数会根据消息的哈希值来选择目标TaskId。
- 第33~36行会根据分组函数返回的是集合还是单个元素而调用不同的函数将结果添加到ret中。
- 第37行返回目标TaskId的集合，消息将被发送到这些Task。

11.4 send-unanchored

Task使用send-unanchored函数来发送消息。该函数会利用tasks-fn函数来获得目标节点的TaskId，然后调用Executuror的:transfer-fn函数将消息发送到发送队列中，最终Worker会利用ZMQ将消息发送给目标Task。若目标Task与当前Task在同一个Worker中，Worker则会将消息直接发送到该Task的接收队列中。send-unanchored的代码如下：

```
1 (defn send-unanchored
2     ([task-data stream values overflow-buffer]
3        (let [^TopologyContext topology-context (:system-context task-data)
4                tasks-fn (:tasks-fn task-data)
5                transfer-fn (-> task-data :executor-data :transfer-fn)
6                out-tuple (TupleImpl. topology-context
7                              values
8                              (.getThisTaskId topology-context)
9                              stream)]
10               (fast-list-iter [t (tasks-fn stream values)]
11               (transfer-fn t
12                  out-tuple
13                  overflow-buffer)
14               )))
15     ([task-data stream values]
16         (send-unanchored task-data stream values nil)
17       ))
```

- 第15~17行是函数的一个重载，为含有3个形参的实现。它将调用基于4个参数的重载，第4个参数表示是否要将发送的消息放于一个缓存里面。对于Ack或Fail消息，系统需要尽快地将其发送出去，因此调用的都是非缓存的重载方法。

- 第2~14行定义了方法的另一个重载。第6~9行构建一个要发送的消息，第10行通过调用tasks-fn函数计算得到该消息的目标TaskId，第11~14行则调用Executor对应的transfer-fn函数将消息发送出去。

11.5　创建 Task

mk-task函数用于创建一个新的Task，其主要目标为通过调用mk-task-data函数为该Task创建对应的数据。该函数将用户定义的钩子函数注册到:user-context数据中，同时向系统组件发送一条消息。目前，系统组件为SystemBolt。不过目前Storm中并没有使用SystemBolt，SystemBolt主要用来统计Worker的运行情况。mk-task的代码如下所示：

```
(defn mk-task [executor-data task-id]
    (let [task-data (mk-task-data executor-data task-id)
          storm-conf (:storm-conf executor-data)]
       (doseq [klass (storm-conf TOPOLOGY-AUTO-TASK-HOOKS)]
          (.addTaskHook ^TopologyContext (:user-context task-data) (-> klass
             Class/forName .newInstance)))
       ;; when this is called, the threads for the executor haven't been started yet,
       ;; so we won't be risking trampling on the single-threaded claim strategy disruptor queue
       (send-unanchored task-data SYSTEM-STREAM-ID ["startup"])
       task-data
       ))
```

11.6　Storm 中传输的消息以及序列化

本节将介绍Storm是如何对消息进行序列化的，并分析实际的传输内容。

KryoTupleSerializer类用于实现对消息的序列化，其代码如下：

```
1  public class KryoTupleSerializer implements ITupleSerializer {
2     KryoValuesSerializer _kryo;
3     SerializationFactory.IdDictionary _ids;
4     Output _kryoOut;
5
6     public KryoTupleSerializer(final Map conf, final GeneralTopologyContext context) {
7        _kryo = new KryoValuesSerializer(conf);
8        _kryoOut = new Output(2000, 2000000000);
9        _ids = new SerializationFactory.IdDictionary(context.getRawTopology());
10    }
11
12    public byte[] serialize(Tuple tuple) {
13       try {
14
15           _kryoOut.clear();
16           _kryoOut.writeInt(tuple.getSourceTask(), true);
17           _kryoOut.writeInt(_ids.getStreamId(tuple.getSourceComponent(),
                  tuple.getSourceStreamId()), true);
18           tuple.getMessageId().serialize(_kryoOut);
```

```
19           _kryo.serializeInto(tuple.getValues(), _kryoOut);
20           return _kryoOut.toBytes();
21       } catch (IOException e) {
22          throw new RuntimeException(e);
23       }
24    }
25 }
```

从第16~19行代码可以看出，Storm实际传输的消息为：

```
SourceTaskId:int；StreamId:int; messageId:MessageId;TupleValues：List<Object>
```

其中，流序号是对组件中所有的流排序后确定的统一编号，IdDictionary类会根据输入的Topology定义获取所有的组件名字以及它们的流序号，然后进行编号。TupleValues为用户实际发送的消息。

MessageId类存储从消息的RootId到其衍生消息的消息ID的异或值的映射关系，它用于跟踪消息，内含一个哈希表。

MessageId类中用于实现序列化的方法如下：

```
1 public void serialize(Output out) throws IOException {
2    out.writeInt(_anchorsToIds.size(), true);
3    for(Entry<Long, Long> anchorToId: _anchorsToIds.entrySet()) {
4       out.writeLong(anchorToId.getKey());
5       out.writeLong(anchorToId.getValue());
6    }
7 }
```

序列化的数据为<元素个数，[RootId, AckVal]+>，而反序列化方法与其类似。

KryoTupleDeserializer类用来对收到的消息字节数组进行反序列化，相关代码如下：

```
public Tuple deserialize(byte[] ser) {
    try {
        _kryoInput.setBuffer(ser);
        int taskId = _kryoInput.readInt(true);
        int streamId = _kryoInput.readInt(true);
        String componentName = _context.getComponentId(taskId);
        String streamName = _ids.getStreamName(componentName, streamId);
        MessageId id = MessageId.deserialize(_kryoInput);
        List<Object> values = _kryo.deserializeFrom(_kryoInput);
        return new TupleImpl(_context, values, taskId, streamName, id);
    }catch(IOException e) {
        throw new RuntimeException(e);
    }
}
```

反序列化的过程是：首先依次读取TaskId，并通过读取的流序号查找映射表获取流名称，接下来调用MessageId反序列化方法读取MessageId信息，然后读取用户发送的消息，最后构建TupleImpl对象并返回。注意这里的流序号只用于消息的传输，其他地方看到的流序号实际上对应于这里的流名称。

第 12 章

Storm的Ack框架

在Storm中，如何表示一条消息是否已被成功处理了呢？Storm采用创新的Ack框架（详情可参照http://github.com/nathanmarz/storm/wiki/Guaranteeing-message-processing）对Topology中的消息进行跟踪。

Storm利用Acker Bolt节点跟踪消息。当Spout发送出去的消息以及这些消息所衍生出来的消息均被成功处理后，Spout将收到对应于该消息的Ack。Ack框架的实现要点如下。

- Storm中每条发送出去的消息都会对应一个随机的消息ID。
- Spout发送消息后，将向Acker Bolt发送一条消息，该消息的内容为<RootId,消息ID>，Acker Bolt将为该消息创建一条跟踪项。
- Bolt产生要发送的消息时，会计算每条新消息的消息ID，并将消息ID发送至Acker Bolt，Acker Bolt对消息ID进行异或后存储。于是，Storm对新发送的消息进行了跟踪。
- Bolt对输入的消息进行Ack时，也会将该消息ID发送到Acker Bolt。Acker Bolt对消息ID进行异或后存储。由于该消息在被发送时，已经向Acker Bolt发送过消息ID，之后在被Ack时又再次发送该消息ID。根据异或的语义，这相当于对该消息的跟踪结束。
- Acker Bolt在更新某一个消息的跟踪值时，若发现其跟踪值变为零，则向Spout节点发送消息，表明Spout发送的这条消息已经被成功处理。
- 所有消息的消息ID都将更新至根消息上，根消息为Spout发送出去的消息，Bolt新产生的消息并不会被单独跟踪。
- 若Spout在发送消息时未指定用于消息跟踪的ID，系统则不对消息进行跟踪。若系统中不含有Acker Bolt，消息也不会被跟踪。
- Spout的每条消息以及由该消息演化出来的所有消息的跟踪负载为16个字节，8个字节的根消息ID以及8个字节的消息跟踪值AckValue。但是，由于Storm中采用HashMap对其进行存储，在32位的JVM中，每条消息至少需要20个字节的额外负载，故一条消息的跟踪需要40个字节左右的负载。许多文档认为只需要20个字节是不正确的。读者可以参考JDK中有关HashMap实现的内容来分析哈希表中每一条消息需要的负载。

12.1　Acker Bolt 的实现分析

Acker Bolt属于系统级的组件,它们由系统创建，主要用来跟踪消息以及该消息衍生出来的消息是否被完全处理。

一条消息从Spout产生之后，经过若干处理节点，可能会衍生出很多消息，而这些消息以Spout发送的消息为根，这就构成一张图结构。在Storm中，只有当所有衍生出来的消息都被成功处理后，这条从Spout发送出来的消息才被认为是成功处理的。Storm中的Acker Bolt就是用来跟踪消息的系统节点。当一个Bolt同时从两种类型的Spout接收消息时，其产生的消息的根为这两个Spout发送的消息，12.3节会对这种情况进行讨论。

mk-acker-bolt函数用来定义一个Acker-bolt，该Bolt实例化自IBolt接口，其代码如下：

```
1 (defn mk-acker-bolt []
2     (let [output-collector (MutableObject.)
3               pending (MutableObject.)]
4        (reify IBolt
5           (^void prepare [this ^Map storm-conf ^TopologyContext context ^OutputCollector collector]
6              (.setObject output-collector collector)
7              (.setObject pending (RotatingMap. 2))
8              )
9           (^void execute [this ^Tuple tuple]
10                  (let [^RotatingMap pending (.getObject pending)
11                           stream-id (.getSourceStreamId tuple)]
12                     (if (= stream-id Constants/SYSTEM_TICK_STREAM_ID)
13                         (.rotate pending)
14                         (let [id (.getValue tuple 0)
15                                   ^OutputCollector output-collector (.getObject
                                         output-collector)
16                                   curr (.get pending id)
17                                   curr (condp = stream-id
18                                            ACKER-INIT-STREAM-ID (-> curr
19                                                                     (update-ack (.getValue
                                                                         tuple 1))
20                                                                 (assoc :spout-task (.getValue
                                                                     tuple 2)))
21                                            ACKER-ACK-STREAM-ID (update-ack curr (.getValue
                                                tuple 1))
22                               ACKER-FAIL-STREAM-ID (assoc curr :failed true))]
23                            (.put pending id curr)
24                            (when (and curr (:spout-task curr))
25                               (cond (= 0 (:val curr))
26                                  (do
27                                     (.remove pending id)
28                                     (acker-emit-direct output-collector
29                                        (:spout-task curr)
30                                        ACKER-ACK-STREAM-ID
31                                        [id]
32                                        ))
33                                  (:failed curr)
34                                  (do
35                                        (.remove pending id)
```

```
36                                    (acker-emit-direct output-collector
37                                        (:spout-task curr)
38                                        ACKER-FAIL-STREAM-ID
39                                        [id]
40                                        ))
41                              ))
42                        (.ack output-collector tuple)
43                        ))))
44          (^void cleanup [this]
45          )
46          )))
```

- 第2~3行定义了Acker Bolt中的两个类成员变量：output-collector用于向外发送数据；pending为RotatingMap类型，用来存储每一个从Spout发送出来的消息的RootId以及目前该消息的跟踪值（AckValue）。Acker Bolt就是通过不断更新和检测跟踪值来判断该消息是否已经被完全成功处理的。RotatingMap主要用于消息的超时。
- 第5~8行定义了Acker Bolt的prepare方法来设置类的成员变量。这里设置pending的桶数目为2。
- 9~46行定义Acker Bolt的execute方法，其中Acker Bolt的输入流有如下4种。

  (1) SYSTEM_TICK_STREAM_ID：系统预定义流，用于消息超时。

  (2) ACKER-INIT-STREAM-ID：Acker Bolt初始化流，由Spout向其发送消息。

  (3) ACKER-ACK-STREAM-ID：Acker Bolt的 Ack消息流，由Bolt向其发送消息。

  (4) ACKER-FAIL-STREAM-ID：Acker Bolt的Fail消息流，由Bolt向其发送消息。

  Acker Bolt首先检测输入消息的来源流，然后根据流的类型进行下列操作。

  - SYSTEM_TICK_STREAM_ID：这种情况下，Acker Bolt会对成员变量pending进行旋转操作，然后退出execute方法，该操作将pending中最早的一个桶中的数据删除掉，于是实现了消息的超时。由于初始化RotatingMap时，未传入关于expire的回调方法，故该操作只是进行简单的删除。如果继续对已经删除掉的消息的RootId进行Ack操作，就会创建新的<RootId,跟踪值>对，但是由于数据已被删除过的原因，跟踪值基本上不会再回到零，所以Spout将永远也收不到它发送出去的这条消息的Ack。Spout会通过自有的超时机制，将这条消息标记为处理失败，然后调用Spout的失败函数来决定对失败消息进行重传还是忽略。这个操作的结果是去除处于僵死状态的消息跟踪。例如，Spout发送出去一条消息A，它需要经过Bolt A和Bolt B，然而Bolt B在处理过程中由于机器故障死掉了，那么消息A对应的<RootId,AckValue>在Acker Bolt中就处于僵死状态。
  - ACKER-INIT-STREAM-ID：此时输入消息的模式为<RootId,RawAckValue,SpoutTaskId>。Acker Bolt会根据RootId取出<RootId, AckValue>，如果不存在RootId，那么AckValue将是RawAckValue。在第18~20行中，Acker Bolt根据输入消息的AckValue对pending中AckValue进行更新，同时将Spout的TaskId作为关键字存储在curr对象中（利用Clojure的assoc方法）。
  - ACKER-ACK-STREAM-ID：输入消息的模式为<RootId,AckValue>，与原有AckValue进行异或操作并存储。

  - ■ ACKER-FAIL-STREAM-ID：输入消息的模式为<RootId>。设置failed为true，表示消息的处理已经失败。
- ❑ 在第25~32行中，若此时消息对应的跟踪值已经为零，那么Storm认为该消息以及所有衍生的消息都已被成功处理，这时会通过向ACK-STREAM流向Spout节点发送消息，模式为<RootId>。
- ❑ 在第36~41行中，若此时消息被标记为失败，那么Storm会通过FAIL-STREAM流向Spout发送消息，模式为<RootId>。
- ❑ 第42行对输入的消息进行Ack操作。实际上这行代码并没有实际的作用，因为Acker收到的消息都是没有进行跟踪的。

12.2　启动消息跟踪

在Spout向外发送消息时，Storm根据系统中是否存在Acker Bolt以及发送的消息是否带有消息序号（MessageId）来决定是否对消息进行跟踪。若消息被跟踪，Acker Bolt将收到一条初始化消息。Spout使用send-spout-msg函数来向外发送消息。该函数在第10章中讨论过，本节主要关心其中与消息跟踪相关的部分，其代码如下：

```
 1 send-spout-msg (fn [out-stream-id values message-id out-task-id]
 2     (.increment emitted-count)
 3     (let [out-tasks (if out-task-id
 4                        (tasks-fn out-task-id out-stream-id values)
 5                        (tasks-fn out-stream-id values))
 6            rooted? (and message-id has-ackers?)
 7            root-id (if rooted? (MessageId/generateId rand))
 8            out-ids (fast-list-for [t out-tasks] (if rooted? (MessageId/generateId rand)))]
 9         (fast-list-iter [out-task out-tasks id out-ids]
10            (let [tuple-id (if rooted?
11                               (MessageId/makeRootId root-id id)
12                               (MessageId/makeUnanchored))
13                     out-tuple (TupleImpl. worker-context
14                                    values
15                                    task-id
16                                    out-stream-id
17                                    tuple-id)]
18                 (transfer-fn out-task
19                     out-tuple
20                     overflow-buffer)
21                 ))
22          (if rooted?
23              (do
24                   (.put pending root-id [task-id
25                                          message-id
26                                          {:stream out-stream-id :values values}
27                                          (if (sampler) (System/currentTimeMillis))])
28                   (task/send-unanchored task-data
29                       ACKER-INIT-STREAM-ID
```

```
30                     [root-id (bit-xor-vals out-ids) task-id]
31                     overflow-buffer))
32             (when message-id
33                 (ack-spout-msg executor-data task-data message-id
34                     {:stream out-stream-id :values values}
35                     (if (sampler) 0))))
36         (or out-tasks [])
37         ))
```

- 第3~8行用于计算得到该消息的接收端Task集合。若系统中含有Acker Bolt，并且Spout在发送消息时指定了MessageId，Storm将对这条消息进行跟踪，并为其生成一条RootId，然后为发送到每一个Task上面的消息也生成一个消息ID。消息ID是通过调用`MessageId`的`generateId`方法来产生的，为一个长整型随机数。
- 第10~21行为每一个接收端Task构建一条消息并将其发送出去。第10~12行会创建`tuple-id`对象，它是一个哈希表，用来表示产生该消息的最初父节点以及对应的Ack值。例如，一个用于连接两个流的Bolt节点会根据输入的两条消息产生一条消息，而产生出来的消息所对应的`tuple-id`域会含有两条记录，分别为每个流的消息源消息的RootId以及目前的跟踪值。对于Spout来讲，RootId即为第7行创建的随机数。若不进行消息跟踪，`tuple-id`则对应一个空的哈希表。
- 第24~27行将`<RootId,元数据>`存储在RotatingMap中。消息元数据中含有Spout的TaskId、MessageId、StreamId以及消息的内容。存储原始的消息内容可以方便以后的失败重传，不过目前Storm中并没有利用已经存储的消息内容，而是希望用户可以通过MessageId来自己维护已经发送出去的消息。当Ack框架判定消息处理失败时，将调用Spout的Fail方法，但Spout的`fail`方法的原型如下所示，其参数仅仅含有`msgId`，storm未将消息内容传回给`fail`方法，或用户无法得到storm缓存的消息。由于Spout发送出去的消息是在内存中的，它没办法保证Spout重启并且消息处理失败后，是否还可以将原始消息返回给用户的回调函数：

```
void fail(Object msgId);
```

- 第28~31行向`ACKER-INIT-STREAM`发送一条消息，该消息为`<RootId, T1^T2^…^Tn, Spout-Task-Id>`，即输入消息RootId为键，发送到每一个Task上面的消息的消息ID的异或值作为初始化的跟踪值，同时需要传入该Spout的TaskId。当Spout收到针对某条消息的Ack或Fail时，可以用该值来验证这条消息是不是由该Spout发送出去的。
- 第32~35行是针对Spout发送消息时带有MessageId但系统中并没有Acker Bolt情况的一种特殊处理。此时，Storm将直接调用Spout的Ack方法，系统不对消息进行跟踪。

12.3 消息跟踪

Bolt在发送消息时，系统需要继续对其进行跟踪，这些由Bolt新发送的消息对应于从Spout收到消息的衍生消息。Bolt使用`bolt-emit`函数来发送消息，本节主要讨论其中与消息跟踪相关的部

分，相关代码如下：

```
1 bolt-emit (fn [stream anchors values task]
2   (let [out-tasks (if task
3           (tasks-fn task stream values)
4           (tasks-fn stream values))]
5       (fast-list-iter [t out-tasks]
6           (let [anchors-to-ids (HashMap.)]
7              (fast-list-iter [^TupleImpl a anchors]
8                 (let [root-ids (-> a .getMessageId .getAnchorsToIds .keySet)]
9                     (when (pos? (count root-ids))
10                       (let [edge-id (MessageId/generateId rand)]
11                           (.updateAckVal a edge-id)
12                           (fast-list-iter [root-id root-ids]
13                              (put-xor! anchors-to-ids root-id edge-id))
14                           ))))
15             (transfer-fn t
16                 (TupleImpl. worker-context
17                 values
18                 task-id
19                 stream
20                 (MessageId/makeId anchors-to-ids)))))
21      (or out-tasks [])))
22
```

- bolt-emit函数的传入参数为消息标记（anchors），它对应于该消息的父节点消息。为了保证Ack系统正常工作，用户需要明确其产生的消息是由哪些消息衍生的。
- 第6行创建一张哈希表，用于存储本消息的消息标记。
- 第8~14行对标记消息的跟踪值进行更新。为了减少与Acker Bolt通信的次数，此处进行了优化。标记消息的跟踪值用来表示在该Bolt上由标记消息所衍生出来的消息的异或值。例如，输入消息T_1产生T_2、T_3和T_4，则输入消息T_1的跟踪值为T_2^T_3^T_4，在对消息T_1进行Ack操作时，发送给Acker Bolt的值为<RootId, T_2^T_3^T_4>。
- 第11行对标记消息的跟踪值进行更新。
- 第12~13行为新产生的消息创建标记，其值为<RootId, edge-id>。

下面看一下在Bolt的OutputCollector中Ack的方法实现：

```
1 (^void ack [this ^Tuple tuple]
2     (let [^TupleImpl tuple tuple
3              ack-val (.getAckVal tuple)]
4         (fast-map-iter [[root id] (.. tuple getMessageId getAnchorsToIds)]
5            (task/send-unanchored task-data
6               ACKER-ACK-STREAM-ID
7             [root (bit-xor id ack-val)])
8           ))
```

对一条消息进行Ack操作时，需要对该消息所有的标记消息进行操作。消息标记表示这条消息的父消息。

fast-map-iter用于遍历一个哈希表。类似地，fast-list-iter用于遍历一个列表，是Storm的Clojure代码中常见的一个函数，它与通常的序列遍历操作是相同的，但理论上效率更高。由于Clojure中的序列都是Lazy-Evaluation（数据真正被用到时产生），为了提高效率，fast-list-iter将其分成了多个部分来弥补Lazy-Evaluation带来的效率损失。我在本机上进行了一些实验，发现fast-list-iter并非如想象中那样加快了速度。例如，在下面的例子中，doseq函数的性能更好：

```
user=> (time (fast-list-iter [a (range 10000000)] ()))
"Elapsed time: 388.642758 msecs"
user=> (time (doseq [item (range 10000000)]()))
"Elapsed time: 175.036754 msecs"
```

Bolt所对应的Fail方法实现起来较为简单，它只是向ACKER-FAIL-STREAM-ID发送了消息的RootId，而Acker Bolt会对该消息进行失败处理，相关代码如下：

```
(^void fail [this ^Tuple tuple]
    (fast-list-iter [root (.. tuple getMessageId getAnchors)]
       (task/send-unanchored task-data
           ACKER-FAIL-STREAM-ID
           [root]))
```

12.4　Ack 机制的例子

图12-1演示了Ack框架是如何工作的。

下面简要解释一下图12-1。

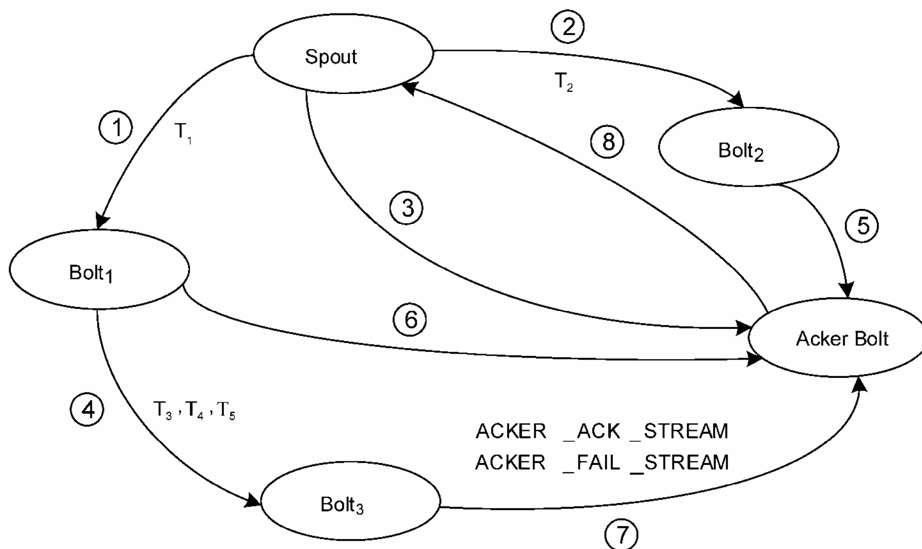


图12-1　Ack框架的例子

- 在第①~②步中，Spout发送T_1到$Bolt_1$，发送T_2到$Bolt_2$。T_1和T_2具有相同的内容，但表示不同的备份。每条消息都会对应一个ID。消息T_1的anchors为<RootId, T_1>，消息T_2的anchors为<RootId, T_2>。
- 在第③步中，Spout在Acker Bolt中注册了一条记录RootId=T_1^T_2。
- 在第④步与第⑥步中，$Bolt_1$发送新的消息T_3、T_4、T_5到$Bolt_3$，同时对输入的消息进行Ack操作，消息的内容为<RootId, T_1^T_3^T_4^T_5>。此时，Acker Bolt中的跟踪项为<RootId, T_1^T_2^T_1^T_3^T_4^T_5=>T_2^T_3^T_4^T_5>。消息T_3的消息标记为<RootId, T_3>。
- 在第⑤步中，$Bolt_2$对输入的消息 T_2进行Ack操作，它没有产生新消息。发送到Acker Bolt的消息为<RootId, T_2>，此时Acker Bolt中的跟踪项为<RootId= T_2^T_3^T_4^T_5^T_2= T_3^T_4^T_5> T_2异或后消失。
- 在第⑦步中，$Bolt_3$对输入的消息进行Ack操作，发送的消息为<RootId, T_3^T_4^T_5>，此时Acker Bolt中的跟踪项为<RootId, T_3^T_4^T_5^ T_3^T_4^T_5=>0>。
- 在第⑧步中，Acker Bolt发现RootId对应的值为零，它认为该RootId对应的消息以及所有衍生出来的消息均已经被成功处理，于是它向Spout发送消息，而Spout将调用Ack回调方法。

注意，第④~⑦步是异步执行的。为了便于讨论，这里给定了顺序。但是，通常对输入消息进行Ack操作是在该Bolt已经发送消息之后进行的。若不想继续对消息进行跟踪，则可以先对输入消息进行Ack操作，然后再发送消息。

每条被处理的消息必须进行Ack或者Fail操作否则，虽然有超时机制可以对过期消息进行清空，但可能导致消息不断重传。

第 13 章 系统运行统计

对于运行中的Topology，Storm会对其运行状态进行统计，并将统计结果展示在Storm UI上。例如，每个Executor发送消息的条数以及被Ack/Fail的消息条数，等等。这些统计信息会被Worker以心跳的方式，通过ZooKeeper这一中间媒介，最终传输给Nimbus，并在 Storm UI上展示。

Storm采用滑动窗口算法来更新统计信息，该算法可以及时反映系统当前的运行状态。我们则可以根据最近十分钟的运行统计来判断Topology是否状态良好。例如，我们可以及时获悉Topology中是否不断有消息产生这些消息是否已被处理等。为了降低运行统计的负载，Storm会使用采样算法来更新统计信息。

本章将对Storm中的运行统计算法以及Storm内置的运行统计进行详细介绍。

13.1 基础数据结构以及更新算法

Storm的运行统计数据被分成了多种类别，例如，每一个Task所发送的消息数目就属于其中一类运行统计数据。对于每一个统计类别，Storm又进一步将其分为三个统计时间区间，即最近10分钟、最近3个小时和最近 1 天。对于每种类别的每一种统计时间区间，Storm都采用了滑动窗口算法来对其进行更新。

例如，对于最近10分钟的运行统计类别，Storm将以30秒为一个时间窗口，即10分钟的区间被分成了20个小的统计窗口。对于新产生的统计信息，若当前时间距离最近的一个时间窗口的创建时间起过了30秒钟，则创建一个新的时间窗口，否则将统计信息更新至最近的时间窗口中。查询运行统计时，将返回最近20个统计窗口的运行统计合并后的结果，同时会将过期的时间窗口删除，这样就实现了窗口的滑动。

对于最近3小时的运行统计类别，Storm同样将其划分为20个统计时间窗口，即每个统计窗口为540秒。可以看出，最近10分钟的运行统计类别的窗口滑动速度更快，统计结果也更为精确。

下面我们分析一下滑动窗口的数据结构以及更新算法。

13.1.1 滑动窗口的数据结构

滑动窗口里Storm实时运行统计的基础，本节简要介绍一下它的主要数据结构。

1. RollingWindow

RollingWindow记录定义了一个滑动窗口：

```
(defrecord RollingWindow [updater merger extractor bucket-size-secs num-buckets buckets])
```

其成员变量分析如下。

- updater：更新运行统计的回调函数。
- merger：获取数据时将多个窗口的统计数据进行合并的回调函数。
- extractor：获取某一个窗口统计数据的回调函数。
- bucket-size-secs：窗口大小，以秒为单位，该变量表示为统计的最短时间间隔。
- num-buckets：窗口数量。
- buckets：每个窗口的统计数据，为Map类型。

下面介绍的函数用于创建以及操作RollingWindow对象。

rolling-window函数用来生成一个RollingWindow对象，其代码如下：

```
(defn rolling-window [updater merger extractor bucket-size-secs num-buckets]
    (RollingWindow. updater merger extractor bucket-size-secs num-buckets {}))
```

update-rolling-window函数用来更新RollingWindow，其形参依次为滑动窗口rw、当前的时间time-secs和用于统计更新参数args，其中参数args中将包含实际的运行统计数据。该函数的代码如下：

```
1 (defn update-rolling-window
2    ([^RollingWindow rw time-secs & args]
3        ;; this is 2.5x faster than using update-in...
4        (let [time-bucket (curr-time-bucket time-secs (:bucket-size-secs rw))
5                  buckets (:buckets rw)
6                  curr (get buckets time-bucket)
7                  curr (apply (:updater rw) curr args)
8                  ]
9           (assoc rw :buckets (assoc buckets time-bucket curr))
10          )))
```

- 第4行利用curr-time-bucket函数来获得当前时间time-secs所属的时间窗口序号。curr-time-bucket函数的定义如下：

  ```
  (defn curr-time-bucket [^Integer time-secs ^Integer bucket-size-secs]
     (* bucket-size-secs (unchecked-divide-int time-secs bucket-size-secs))
     )
  ```

  而窗口序号是根据当前时间time-secs及bucket-size-secs计算得到的，计算公式如下：

  ```
  #Bucket=bucket-size-secs*(time-secs/bucket-size-secs)
  ```

  可以看出，窗口序号并非从零开始，而是对bucket-size-secs进行取整后得到的时间点。领会这一点对于理解函数功能是非常重要的。

- 第6行根据窗口序号time-bucket从滑动窗口中获得其对应的窗口。
- 第7行调用滑动窗口的updater回调函数，利用更新参数对当前窗口进行更新。通常情况下，若curr为空，则updater函数会创建一个新的窗口。
- 第9行将当前窗口curr与窗口序号time-bucket绑定，然后存储在滑动窗口的buckets变量中。

value-rolling-window函数用于获得滑动窗口所对应的值，通常需要将其中各个时间窗口的统计值进行合并，其代码如下：

```
1 (defn value-rolling-window [^RollingWindow rw]
2    ((:extractor rw)
3       (let [values (vals (:buckets rw))]
4          (apply (:merger rw) values)
5          )))
```

- 第3行获得滑动窗口的buckets哈希表的值集合，并将结果存储到values变量中。
- 第4行利用滑动窗口的merger方法将values中的值合并，然后将结果返回。

cleanup-rolling-window函数用于清除滑动窗口中过期的窗口，其代码如下：

```
1 (defn cleanup-rolling-window [^RollingWindow rw]
2   (let [buckets (:buckets rw)
3            cutoff (- (current-time-secs)
4                       (* (:num-buckets rw)
5                          (:bucket-size-secs rw)))
6            to-remove (filter #(< % cutoff) (keys buckets))
7            buckets (apply dissoc buckets to-remove)]
8     (assoc rw :buckets buckets)
9 ))
```

- 第3~4行计算一个时间点cutoff，该时间点以前的窗口都将被清除掉。其计算方法如下：

```
cutoff = current-time-secs - (num-buckets * bucket-size-secs)
```

- 第5行获得要移除的窗口序号，它是通过将时间点cutoff与窗口序号相比较来得到的。
- 第6行将这些窗口序号所对应的值从哈希表中清除。

rooling-window-size函数用来获得特定滑动窗口所对应的统计时间间隔，计算方法为window-size = bucket-size-secs * num-buckets。该函数的实现如下：

```
(defn rolling-window-size [^RollingWindow rw]
   (* (:bucket-size-secs rw) (:num-buckets rw)))
```

2. RollingWindowSet

RollingWindowSet记录表示一组滑动窗口，这组滑动窗口具有相同的updater和extractor回调方法：

```
(defrecord RollingWindowSet [updater extractor windows all-time])
```

下面介绍的函数用来创建以及更新滑动窗口集合。

rolling-window-set函数用于生成RollingWindowSet记录，它利用dofor来生成一组具有相同窗口数目的滑动窗口，而每个滑动窗口对应于不同的时间窗口，分别是30秒、540秒以及4320秒将不同的时间窗口乘上对应窗口数目（这里是20），则滑动窗口对应的时间间隔分别是10分钟、3小时以及1天。rolling-window-set的代码如下：

```
(defn rolling-window-set [updater merger extractor num-buckets & bucket-sizes]
    (RollingWindowSet. updater extractor (dofor [s bucket-sizes] (rolling-window updater merger
      extractor s num-buckets)) nil)
    )
```

形参num-buckets是由NUM-STAT-BUCKETS变量定义的，默认值为20，即每个滑动窗口含有20个小的统计窗口。类似地，形参bucket-sizes是由STAT-BUCKETS定义的，表示每种类型的滑动窗口的窗口大小。相关代码如下：

```
(def NUM-STAT-BUCKETS 20)
;; 10 minutes, 3 hours, 1 day
(def STAT-BUCKETS [30 540 4320])
```

窗口越小，则更新越频繁，统计数据的准确性也将更高。

update-rolling-window-set函数通过调用update-rolling-window来更新滑动窗口集合中的每一个滑动窗口，其代码如下：

```
1 (defn update-rolling-window-set
2     ([^RollingWindowSet rws & args]
3         (let [now (current-time-secs)
4               new-windows (dofor [w (:windows rws)]
5                                (apply update-rolling-window w now args))]
6           (assoc rws :windows new-windows :all-time (apply (:updater rws) (:all-time rws) args))
7           )))
```

注意在第6行中，会调用滑动窗口集合rws的更新回调函数updater对rws的all-time关键字进行更新。all-time关键字对应于整个滑动窗口的总值。

cleanup-rolling-window-set函数通过调用cleanup-rolling-window清理每一个滑动窗口集合中过期的时间窗口，其代码如下：

```
(defn cleanup-rolling-window-set
    ([^RollingWindowSet rws]
        (let [windows (:windows rws)]
            (assoc rws :windows (map cleanup-rolling-window windows))
            )))
```

value-rolling-window-set函数用于获取每一个滑动窗口的当前值，它返回一个哈希表，其中键为当前滑动窗口的窗口大小（例如10分钟），值为通过调用value-rolling-window函数计算得到的值。其代码如下：

```
(defn value-rolling-window-set [^RollingWindowSet rws]
```

```
(merge
    (into {}
       (for [w (:windows rws)]
          {(rolling-window-size w) (value-rolling-window w)}
          ))
    {:all-time ((:extractor rws) (:all-time rws))}))
```

13.1.2 滑动窗口的回调函数

滑动窗口中存储的统计信息可以为基本数值类型或平均值类型，它们分别对应着不同的计算方法，具体如下所示。

1. 更新函数

Storm中定义了两种类型的更新函数：一种用于更新基本数值，另外一种用于更新平均值。

incr-val函数用于对值进行累加，该函数有两个重载，默认的累加值为1。amap用来存储运行统计结果，若amap中没有对应的键，则将默认累加值初始化为0，这通常对应于新生成一个统计窗口的情况。该函数的代码如下：

```
(defn- incr-val
    ([amap key]
        (incr-val amap key 1))
    ([amap key amt]
        (let [val (get amap key (long 0))]
           (assoc amap key (+ val amt))
           )))
```

incr-val函数的第二个重载用于传入一个amt值，表示增加量。在Storm中，该值为一个根据采样频率估得的值。例如若Storm中采样频率为5%，则该值为20，即1次更新将相当于20次更新。Storm中采用even-sampler采样算法，详情请参考第4章。

update-avg函数用来更新平均值。值curr中的第一部分为累加值，第二部分为消息的数目。update-keyed-avg函数用于更新哈希表中所有的值。相关代码如下：

```
(defn- update-avg [curr val]
    (if curr
        [(+ (first curr) val) (inc (second curr))]
        [val (long 1)]
        ))
(defn- update-keyed-avg [amap key val]
    (assoc amap key (update-avg (get amap key) val)))
```

2. 合并函数

基本值类型的合并函数比较简单，Storm并没有额外定义一个函数，而是定义了一个局部函数。Clojure的关键字partial用于定义一个局部函数。基本值类型的合并函数利用Clojure内置的merge-with函数进行累加。该函数的定义如下：

```
(partial merge-with +)
```

merge-avg函数用来计算合并后的平均值。变量avg中的第一部分为所有的分子第二部分为所有的分母，算法为将分子与分母分别累加。merge-keyed-avg函数对哈希表vals中每一个值对进行计算。相关代码如下：

```
(defn- merge-avg [& avg]
    [(apply + (map first avg))
       (apply + (map second avg))
       ])
(defn- merge-keyed-avg [& vals]
    (apply merge-with merge-avg vals))
```

3. 提取函数

extract-avg函数用来计算平均值，extract-keyed-avg函数则对哈希表vals中的每一个值对进行计算，相关代码如下：

```
(defn- extract-avg [pair]
    (double (/ (first pair) (second pair))))
(defn- extract-keyed-avg [vals]
    (map-val extract-avg vals))
```

基本值类型的提取器则比较简单，直接返回即可，其代码如下：

```
(defn- counter-extract [v]
    (if v v {}))
```

13.1.3 滑动窗口集合的类型

根据存储统计信息类型的不同，Storm中主要含有基本值类型和平均值类型这两种滑动窗口集合类型。基本值类型的滑动窗口集合可用来统计发送的消息总数，即平均值类型的滑动窗口集合则可用来统计发送消息的平均时延。

keyed-counter-rolling-window-set函数用于定义基本值类型的滑动窗口集合，其代码如下：

```
(defn keyed-counter-rolling-window-set [num-buckets & bucket-sizes]
    (apply rolling-window-set incr-val (partial merge-with +) counter-extract num-buckets
        bucket-sizes))
```

avg-rolling-window-set函数用于定义平均值类型的滑动窗口集合，其代码如下：

```
(defn avg-rolling-window-set [num-buckets & bucket-sizes]
    (apply rolling-window-set update-avg merge-avg extract-avg num-buckets bucket-sizes)
    )
```

keyed-avg-rolling-window-set函数用于定义哈希表上的平均值类型的滑动窗口集合，其代码如下：

```
(defn keyed-avg-rolling-window-set [num-buckets & bucket-sizes]
    (apply rolling-window-set update-keyed-avg merge-keyed-avg extract-keyed-avg num-buckets
        bucket-sizes))
```

这几种滑动窗口集合的本质区别在于其滑动窗口中存储数据的类型不同，并且它们需要采用不同的回调函数进行操作，具体情况如表13-1所示。

表13-1　滑动窗口集合的类型

	keyed-counter-rolling-window-set	avg-rolling-window-set	keyed-avg-rolling-window-set
统计窗口中存储的数据类型	哈希表 值为简单类型	一个<Key, Value>值对 键为Sum 值为Count Avg=Sum/Count	哈希表 值为<Sum, Count>对
更新函数	incr-val	update-avg	update-keyed-avg
合并函数	(partial merge-with +)	merge-avg	merge-keyed-avg
提取函数	counter-extract	extract-avg	extract-keyed-avg

在许多函数中，可以对关键字：all-time进行操作，例如update-rolling-window-set是滑动窗口的含有的统计汇总，其数据类型与小统计窗口相同。

在Storm中，我们主要采用keyed-counter-rolling-window-set以及keyed-avg-rolling-window-set类型的滑动窗口集合，统计窗口中哈希表的键为组件的流号（StreamId）。

13.2　Storm 中的统计信息

本节将分析Storm中含有的统计信息类别及其更新时机，这对于理解Storm UI上所展示的内容至关重要。

13.2.1　Stats中定义的统计类别

在Storm中，每种类别的统计都对应着一个滑动时间窗口集合对象，具体如表13-2所示。

表13-2　Storm中的统计类别

统计名称	类　别	类　型	更新函数
:emitted	COMMON-FIELDS	keyed-counter-rolling-window-set	emitted-tuple!
:transferred	COMMON-FIELDS	keyed-counter-rolling-window-set	transferred-tuples!
:acked	BOLT	keyed-counter-rolling-window-set	bolt-acked-tuple!
:failed	BOLT	keyed-counter-rolling-window-set	bolt-failed-tuple!
:process-latencies	BOLT	keyed-avg-rolling-window-set	bolt-acked-tuple!
:executed	BOLT	keyed-counter-rolling-window-set	bolt-execute-tuple!
:execute-latencies	BOLT	keyed-avg-rolling-window-set	bolt-execute-tuple!
:acked	SPOUT	keyed-counter-rolling-window-set	spout-acked-tuple!
:failed	SPOUT	keyed-counter-rolling-window-set	spout-failed-tuple!
:complete-latencies	SPOUT	keyed-avg-rolling-window-set	spout-acked-tuple!

其中COMMON-FIELDS表示Spout和Bolt都具有的统计类别。详情可参考源代码中mk-spout-stats以及make-bolt-stats函数的定义。

13.2.2 运行统计的更新

Storm中定义了一个宏来更新这些统计信息，该宏的含义为，从stats对象中根据path定义的关键字来获取其对应的滑动窗口集合，然后传入args参数，再调用update-rolling-window-set方法对滑动窗口集合进行更新，其中stats对象为Storm中的滑动窗口集合，path为统计类别（例如:acked）。下面的宏可以完成此功能：

```
(defmacro update-executor-stat! [stats path & args]
    (let [path (collectify path)]
        (swap! (-> ~stats ~@path) update-rolling-window-set ~@args)
        ))
```

在Storm中，我们定义了诸如spout-acked-tuple!的函数来更新滑动窗口集合。Storm将在合适的时间点去调用这些函数。

13.2.3 运行统计的更新时间点

运行统计对于Storm来讲是至关重要的，但过多的运行统计操作会降低系统的吞吐量。在Storm中，我们采用采样的方式并根据这部分统计的结果来估计全局统计结果。在Storm中，UI上反映出来的统计信息为全局统计结果，不过它是根据采样结果估计得到的。在本节中，我们介绍一下Storm是如何在系统中嵌入这些运行统计代码的。

Storm定义了一些采样函数，相关代码如下：

```
:sampler (mk-stats-sampler storm-conf)
execute-sampler (mk-stats-sampler (:storm-conf executor-data))
emit-sampler (mk-stats-sampler storm-conf)
```

其中变量：sampler是Executor的数据变量，变量execute-sampler在Bolt的主线程循环中定义，而emit-sampler是在Task的数据变量。这些采样函数的实例将用于不同的统计目的。mk-stats-sampler函数将产生一个均匀采样器实例（even-sampler）。

本节接下来的部分将对Storm中的运行统计更新函数及其调用时机展开讨论。

1. bolt-execute-tuple!函数

对于Bolt类型的Executor，其tuple-action-fn函数将调用Bolt对象的execute方法。调用execute方法前后是对一些运行统计进行更新的合适时机。下面具体来看一下这些运行统计代码是如何嵌入到系统中的：

```
1 (let [task-data (get task-datas task-id)
2          ^IBolt bolt-obj (:object task-data)
3          user-context (:user-context task-data)
4         sampler? (sampler)
5         execute-sampler? (execute-sampler)
6         now (if (or sampler? execute-sampler?) (System/currentTimeMillis))]
7    (when sampler?
8       (.setProcessSampleStartTime tuple now))
```

```
9    (when execute-sampler?
10       (.setExecuteSampleStartTime tuple now))
11   (.execute bolt-obj tuple)
12   (let [delta (tuple-execute-time-delta! tuple)]
13       (task/apply-hooks user-context .boltExecute (BoltExecuteInfo. tuple task-id delta))
14       (when delta
15           (builtin-metrics/bolt-execute-tuple! (:builtin-metrics task-data)
16               executor-stats
17               (.getSourceComponent tuple)
18               (.getSourceStreamId tuple)
19               delta)
20           (stats/bolt-execute-tuple! executor-stats
21               (.getSourceComponent tuple)
22               (.getSourceStreamId tuple)
23               delta)))))))]
```

- 第4行调用采样器sampler来计算本条消息是否被采样，若被采样，将调用输入消息的setProcessSampleStartTime方法设置消息的处理开始时间。消息的处理时间与消息的执行时间是不同的概念，前者表示从收到该消息到该消息进行Ack或Fail操作的时间间隔，而后者则表示整个execute函数的执行时间。通常，这两个值是比较接近的。
- 第5行调用采样函数execute-sampler，来确定本条消息是否被采样。
- 在第9~10行中，若采样则调用输入消息的setExecuteSampleStartTime方法来设置execute函数的调用开始时间。
- 在第12行中，若delta不为空，则表明该消息被采样了。
- 第13行调用用户实现的钩子方法，这与系统的运行统计无关，不过此时用户可以实现钩子方法来获得运行统计信息。
- 第15~19行调用builtin-metrics/bolt-execute-tuple!更新内置统计信息，其中内置的统计信息将在下一章中讨论。
- 第20~23行调用stats/bolt-execute-tuple!函数更新统计信息。

2. bolt-acked-tuple!函数

Bolt类型Executor的bolt-acked-tuple!函数在消息进行Ack操作时调用，以获取Ack的消息数目以及消息的处理时延等统计信息，相关代码如下：

```
1 (^void ack [this ^Tuple tuple]
2     (let [^TupleImpl tuple tuple
3             ack-val (.getAckVal tuple)]
4         (fast-map-iter [[root id] (.. tuple getMessageId getAnchorsToIds)]
5             (task/send-unanchored task-data
6                 ACKER-ACK-STREAM-ID
7                 [root (bit-xor id ack-val)])
8         ))
9     (let [delta (tuple-time-delta! tuple)]
10        (task/apply-hooks user-context .boltAck (BoltAckInfo. tuple task-id delta))
11        (when delta
12           (builtin-metrics/bolt-acked-tuple! (:builtin-metrics task-data)
13               executor-stats
```

```
14              (.getSourceComponent tuple)
15              (.getSourceStreamId tuple)
16              delta)
17          (stats/bolt-acked-tuple! executor-stats
18              (.getSourceComponent tuple)
19              (.getSourceStreamId tuple)
20              delta))))
```

- 第9行调用tuple-time-delta!函数计算消息的处理时间。在Bolt的execute函数中，若消息被sampler，采样器采样，则设定processSampleStartTime。当该消息被Ack时，则判断process SampleStartTime是否为空，若不为空则计算消息的处理时间。可见，消息的处理时间是跨节点计算的，而执行时间是在同一节点中计算得到的。
- 第10~20行调用钩子方法以及运行统计的更新函数更新相应的运行统计。tuple-time-delta!的定义如下：

```
(defn- tuple-time-delta! [^TupleImpl tuple]
    (let [ms (.getProcessSampleStartTime tuple)]
       (if ms
          (time-delta-ms ms))))
```

3. bolt-failed-tuple!函数

类似地，在Bolt类型Executor的fail方法中，将调用bolt-failed-tuple!更新函数。是否进行运行统计更新同样取决于采样器sampler。下面的代码演示了fail方法是如向嵌入运行统计代码的：

```
(^void fail [this ^Tuple tuple]
    (fast-list-iter [root (.. tuple getMessageId getAnchors)]
        (task/send-unanchored task-data
            ACKER-FAIL-STREAM-ID
            [root]))
    (let [delta (tuple-time-delta! tuple)]
        (task/apply-hooks user-context .boltFail (BoltFailInfo. tuple task-id delta))
        (when delta
            (builtin-metrics/bolt-failed-tuple! (:builtin-metrics task-data)
                executor-stats
                (.getSourceComponent tuple)
                (.getSourceStreamId tuple))
             (stats/bolt-failed-tuple! executor-stats
                 (.getSourceComponent tuple)
                 (.getSourceStreamId tuple)
                 delta))))
```

4. transferrred-tuple!和emitted-tuple!函数

这两个函数主要在Task的发送函数中调用，Spout或Bolt的Executor在发送消息时都会通过Task的发送函数来实现，相关代码如下：

```
1 (fn ([^Integer out-task-id ^String stream ^List values]
2          (when debug?
3             (log-message "Emitting direct: " out-task-id "; " component-id " " stream " " values))
4          (let [target-component (.getComponentId worker-context out-task-id)
```

```
5                    component->grouping (get stream->component->grouper stream)
6                    grouping (get component->grouping target-component)
7                    out-task-id (if grouping out-task-id)]
8              (when (and (not-nil? grouping) (not= :direct grouping))
9                 (throw (IllegalArgumentException. "Cannot emitDirect to a task expecting a regular
                     grouping")))
10             (apply-hooks user-context .emit (EmitInfo. values stream task-id [out-task-id]))
11             (when (emit-sampler)
12                (builtin-metrics/emitted-tuple! (:builtin-metrics task-data) executor-stats
                     stream)
13                (stats/emitted-tuple! executor-stats stream)
14                (if out-task-id
15                    (stats/transferred-tuples! executor-stats stream 1)
16                    (builtin-metrics/transferred-tuple! (:builtin-metrics task-data)
                         executor-stats stream 1)))
17             (if out-task-id [out-task-id])
18             ))
19    ([^String stream ^List values]
20            (when debug?
21               (log-message "Emitting: " component-id " " stream " " values))
22            (let [out-tasks (ArrayList.)]
23               (fast-map-iter [[out-component grouper] (get stream->component->grouper stream)]
24                   (when (= :direct grouper)
25                       ;;  TODO: this is wrong, need to check how the stream was declared
26                       (throw (IllegalArgumentException. "Cannot do regular emit to direct
                             stream")))
27                   (let [comp-tasks (grouper task-id values)]
28                       (if (or (sequential? comp-tasks) (instance? Collection comp-tasks))
29                          (.addAll out-tasks comp-tasks)
30                          (.add out-tasks comp-tasks)
31                          )))
32              (apply-hooks user-context .emit (EmitInfo. values stream task-id out-tasks))
33              (when (emit-sampler)
34                  (stats/emitted-tuple! executor-stats stream)
35                  (builtin-metrics/emitted-tuple! (:builtin-metrics task-data) executor-stats
                      stream)
36                  (stats/transferred-tuples! executor-stats stream (count out-tasks))
37                  (builtin-metrics/transferred-tuple! (:builtin-metrics task-data) executor-stats
                      stream (count out-tasks)))
38              out-tasks)))
39
```

- 第4~18行对应于向直接流发送消息。在第11行中，若采样器emit-sampler为true，则调用emitted-tuple!更新发送的消息数目；若含有out-task-id，则调用transferred-tuples!传输消息。可以看出，若目标节点没有收听该直接流，则emitted-tuple!会被调用，而transferred-tuple!不会被调用，即该消息实际上并没有被传输，从这一点也可看出这两个统计的差别。
- 第19~38行用于向目标节点集合发送消息。第33~37行用于更新统计信息。若含有多个接收端Task时，则emitted-tuple!函数只算一条消息，而transferred-tuple!则被算作多条消息。传输的消息数目反映了实际传输的消息数目。这也是两个函数定义略有区别的原因。

这两个函数的代码如下：

```
(defn emitted-tuple! [stats stream]
    (update-executor-stat! stats [:common :emitted] stream (stats-rate stats)))

(defn transferred-tuples! [stats stream amt]
    (update-executor-stat! stats [:common :transferred] stream (* (stats-rate stats) amt)))
```

transferred-tuples!函数需要传入amt，它通过计算(stats-rate stats)*amt来获得更新数量，而emitted-tuple!则使用(stats-rate stats)作为更新数量。

5. spout-acked-tuple!和spout-failed-tuple!函数

这两个函数会在Spout节点收到消息的Ack或Fail时被调用，这表明只有当Spout打开消息跟踪时，两类统计信息才会被更新。send-spout-msg将决定一条消息是否被运行统计选中，下面分析该函数的代码：

```
1 (if rooted?
2   (do
3         (.put pending root-id [task-id
4                                message-id
5                                {:stream out-stream-id :values values}
6                                (if (sampler) (System/currentTimeMillis))])
7         (task/send-unanchored task-data
8             ACKER-INIT-STREAM-ID
9             [root-id (bit-xor-vals out-ids) task-id]
10            overflow-buffer))
11  (when message-id
12         (ack-spout-msg executor-data task-data message-id
13             {:stream out-stream-id :values values}
14             (if (sampler) 0))))
15 (or out-tasks [])
```

- ❑ 第6行调用采样器sampler，若返回为ture，则设置为当前时间，否则为空。
- ❑ 在第14行中，当消息被跟踪，但系统却没有Acker Bolt节点时，消息将直接被Spout节点Ack。若该消息被采样器选中，统计数据也会被更新。

接下来分析ack-spout-msg函数，相关代码如下：

```
1 (defn- ack-spout-msg [executor-data task-data msg-id tuple-info time-delta]
2    (let [storm-conf (:storm-conf executor-data)
3            ^ISpout spout (:object task-data)
4             task-id (:task-id task-data)]
5        (when (= true (storm-conf TOPOLOGY-DEBUG))
6          (log-message "Acking message " msg-id))
7        (.ack spout msg-id)
8        (task/apply-hooks (:user-context task-data) .spoutAck (SpoutAckInfo. msg-id task-id
             time-delta))
9        (when time-delta
10       (builtin-metrics/spout-acked-tuple! (:builtin-metrics task-data) (:stats
             executor-data)(:stream tuple-info) time-delta)
11         (stats/spout-acked-tuple! (:stats executor-data) (:stream tuple-info) time-delta))))
```

若输入的time-delta不为空，则调用spout-acked-tuple!来更新统计。time-delta通过下面代

码中的方法得到,若start-time-ms不为空,则用当前时间减去start-time-ms。(变量start-time-ms是在sent-spout-msg函数中根据是否被采样选中而设置的):

```
[time-delta (if start-time-ms (time-delta-ms start-time-ms))]
```

13.2.4　获取统计数据

cleanup-stats!函数用来通知滑动窗口进行滑动，这样过期的统计窗口将被清除，相关代码如下：

```
(defn- cleanup-stat! [stat]
    (swap! stat cleanup-rolling-window-set))
```

Storm在获取统计信息时，会首先调用窗口滑动函数，然后再获取当前的统计值。也就是说，在Storm获取统计信息之前，每个滑动窗口的实际统计窗口数目可能大于预定义的统计窗口数目。cleanup-spout-stats!和cleanup-bolt-stats!函数分别用于清理Spout和Bolt中各个统计信息的滑动窗口集合。

value-bolt-stats!函数用来获取统计信息，相关代码如下：

```
(defn- value-common-stats [^CommonStats stats]
    (merge
        (value-stats stats COMMON-FIELDS)
        {:rate (:rate stats)}))

(defn value-bolt-stats! [^BoltExecutorStats stats]
    (cleanup-bolt-stats! stats)
    (merge (value-common-stats (:common stats))
        (value-stats stats BOLT-FIELDS)
        {:type :bolt}))
```

value-stats函数用来获取每一种统计类型的统计数据，其返回结果是一个哈希表，键为统计类型，值为统计结果。值的类型也为哈希表，表示每个流上的统计值。另外，关键字:type表示节点类型，:rate表示为采样频率。

render-stats!会函数调用value-bolt-stats!及value-spout-stats!来获取统计信息，相关代码如下：

```
(defmulti render-stats! class-selector)

(defmethod render-stats! SpoutExecutorStats [stats]
    (value-spout-stats! stats))

(defmethod render-stats! BoltExecutorStats [stats]
    (value-bolt-stats! stats))
```

Storm的运行统计结果会通过心跳的方式传送到ZooKeeper，而Nimbus从ZooKeeper中获取这些信息并在Storm UI进行展示。具体地，Worker中会调用如下方法：

```
1 (defnk do-executor-heartbeats [worker :executors nil]
2     ;; stats is how we know what executors are assigned to this worker
3     (let [stats (if-not executors
4                     (into {} (map (fn [e] {e nil}) (:executors worker)))
5                     (->> executors
6                       (map (fn [e] {(executor/get-executor-id e) (executor/render-stats e)}))
7                       (apply merge)))
8             zk-hb {:storm-id (:storm-id worker)
9                       :executor-stats stats
10                      :uptime ((:uptime worker))
11                      :time-secs (current-time-secs)
12                      }]
13        ;; do the zookeeper heartbeat
14        (.worker-heartbeat! (:storm-cluster-state worker) (:storm-id worker) (:assignment-id
            worker) (:port worker) zk-hb)
15        ))
```

其中第6行调用每一个Executor的`render-stats`方法来获得该Executor的运行统计信息（这个方法实际上调用了stats.clj中的`render-stats!`方法）。之后，利用统计信息构成Executor的心跳信息发送出去。

13.3 运行统计的 Thrift 结构

各个节点的运行统计结果存储在ZooKeeper中，而Nimbus从ZooKeeper中获得这些结果将其显示在StormUI上。Nimbus采用Thrift结果来存储这些运行统计信息。下面简要介绍这些数据结构。

Storm中定义了Thrift类型的数据结构`BoltStats`和`SpoutStats`，统计信息需要放入这两种类型的对象中。Nimbus会调用相关函数来将这些对象初始化。

`to-global-stream-id`函数根据组件以及流来构建`GlobalStream`对象，其中`GlobalStream`对象为流的全局标识符，其代码如下：

```
(defn to-global-stream-id [[component stream]]
    (GlobalStreamId. component stream))
```

`window-set-converter`函数的主要目的是构建与`GlobalStream`对应的统计值。stats对象的映射关系为`bucketsize->[ComponentId, StreamId]->Value`。下面看一下该函数的实现：

```
(defn window-set-converter
    ([stats key-fn]
     ;; make the first key a string,
     (into {}
         (for [[k v] stats]
             [(str k)
                 (into {}
                    (for [[k2 v2] v]
                       [(key-fn k2) v2]))]
             )
         ))
([stats]
    (window-set-converter stats identity)))
```

`thriftify-specific-stats:bolt`函数为Bolt类型的函数实现，它将Bolt中各种类型的统计信息转化成为Thrift中定义的类型，相关代码如下：

```
(defmethod thriftify-specific-stats :bolt
    [stats]
    (ExecutorSpecificStats/bolt
        (BoltStats. (window-set-converter (:acked stats) to-global-stream-id)
            (window-set-converter (:failed stats) to-global-stream-id)
            (window-set-converter (:process-latencies stats) to-global-stream-id)
            (window-set-converter (:executed stats) to-global-stream-id)
            (window-set-converter (:execute-latencies stats) to-global-stream-id)
            )))
```

第 14 章 系统运行统计的另一种实现

除了上一章介绍的内容外，Storm还提供了另一种运行统计的方法，我们称之为内置的运行统计。未来，Storm可能会采取这种方式收集运行统计信息。这些内置的运行统计与目前正在使用的运行统计类似，它们都包括了诸如每个组件发送的消息数目和处理时延等信息，但内置的运行统计却是通过不同的处理机制才收集到这些统计数据的。总的来说，内置统计信息的收集有如下几个主要步骤。

(1) 创建Task数据时，系统会将内置的统计类别注册到该Task所对应的`TopologyContext`对象中。

(2) 创建Executor时会创建一个计时器，该计时器将定期向流`METRICS_TICK_STREAM_ID`发送统计触发消息（Tick）。

(3) Task收到统计触发消息后，会将Task中注册的统计类别的结果发送到流`METRICS_STREAM_ID`上去。于是，系统中的任何一个组件都会向该流发送统计结果消息。

(4) 用户创建一个类实现`IMetricsConsumer`接口，并将该类注册到Storm中，这需要通过参数`TOPOLOGY_METRICS_CONSUMER_REGISTER`以及类`Config`的`registerMetricsConsumer`来完成。

(5) 当系统中存在已注册的`IMetricsConsumer`类时，Storm将为每一个类创建一个Bolt节点，该Bolt节点负责收听所有组件发送到`METRICS_STREAM_ID`的消息。

(6) 每个Worker都会创建一个SystemBolt，该Bolt节点会将其所在进程的内存统计、Java垃圾回收等信息发送到`METRICS_STREAM_ID`，于是IMetricsConsumer Bolt节点便也可以收到这些消息了。

从上面的步骤可以看出，Storm中可以存在全局的Bolt节点，以收集这些运行统计结果。但目前，Storm并没有添加此类默认的运行统计收集节点。对于本章，读者可以选择性地阅读。

14.1 内置统计信息的计算

内置的运行统计主要有两种类型：一种是计数类型的，如发送的消息数目、被Ack的消息数目等；另外一种是时延类型的，如消息的平均处理时延等。Storm提供了一套较为灵活的API系统，可以帮助用户实现自定义的统计信息，以及将它们注册到`TopologyContext`中。内置统计信息的类关系如图14-1所示。

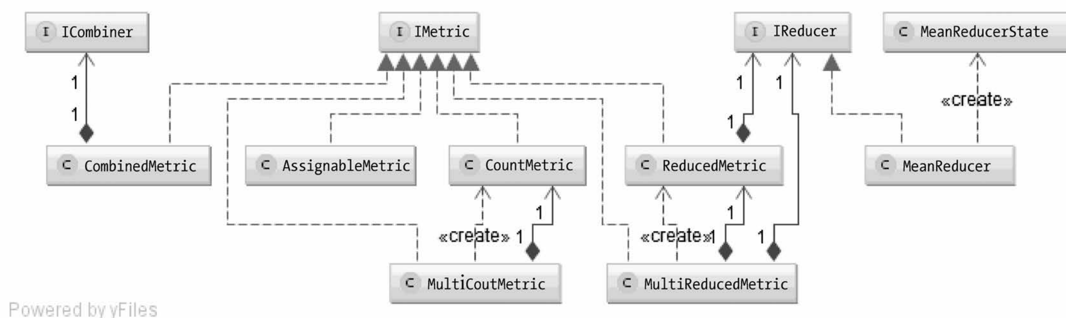


图14-1　内置统计信息的类关系

这些接口较为简单，读者可以参考其源代码进行学习，本节主要对其中几个较为重要的类进行介绍。

14.1.1　MultiCountMetric

CountMetric类内含一个long类型的值，并提供incr和incrBy方法来对该值进行更新。

MultiCountMetric类含有一组CountMetric对象，该类在更新时，会通过调用scope函数来找到相应的CountMetric对象。在Storm中，scope常作为流标识符存在，即ComponentId:StreamId。

例如，在统计Spout发送的消息条数时，由于Spout可以向多个流发送消息，需对它们分开进行统计，因此需要用MultiCountMetric类来处理，这也体现了MultiCountMetric类的设计目标。该类的实现如下：

```
public class MultiCountMetric implements IMetric {
    Map<String, CountMetric> _value = new HashMap();

    public MultiCountMetric() {
    }

    public CountMetric scope(String key) {
        CountMetric val = _value.get(key);
        if(val == null) {
            _value.put(key, val = new CountMetric());
        }
        return val;
    }

    public Object getValueAndReset() {
        Map ret = new HashMap();
        for(Map.Entry<String, CountMetric> e : _value.entrySet()) {
            ret.put(e.getKey(), e.getValue().getValueAndReset());
        }
        return ret;
    }
```

```
}
public class CountMetric implements IMetric {
    long _value = 0;

    public CountMetric() {
    }

    public void incr() {
        _value++;
    }

    public void incrBy(long incrementBy) {
        _value += incrementBy;
    }

    public Object getValueAndReset() {
        long ret = _value;
        _value = 0;
        return ret;
    }
}
```

14.1.2　MultiReducedMetric

MeanReducer类用来计算平均值。例如，在更新消息的平均处理时间时，Storm所获得的信息为单条消息的处理时间，这种情况下便可使用MeanReducer类保存总处理时间以及消息条数，继而方便地计算出平均时间。

MultiReducedMetric类含有一组ReducedMetric对象。目前，Storm采用MeanReducer作为其实现。该类的目标与MultiCountMetric相同，是为了区分同一个组件上不同流的统计结果。该类的实现如下：

```
public class MultiReducedMetric implements IMetric {
    Map<String, ReducedMetric> _value = new HashMap();
    IReducer _reducer;

    public MultiReducedMetric(IReducer reducer) {
        _reducer = reducer;
    }

    public ReducedMetric scope(String key) {
        ReducedMetric val = _value.get(key);
        if(val == null) {
            _value.put(key, val = new ReducedMetric(_reducer));
        }
        return val;
    }

    public Object getValueAndReset() {
        Map ret = new HashMap();
```

```
        for(Map.Entry<String, ReducedMetric> e : _value.entrySet()) {
            Object val = e.getValue().getValueAndReset();
            if(val != null) {
                ret.put(e.getKey(), val);
            }
        }
        return ret;
    }
}

class MeanReducerState {
    public int count = 0;
    public double sum = 0.0;
}

public class MeanReducer implements IReducer<MeanReducerState> {
    public MeanReducerState init() {
        return new MeanReducerState();
    }

    public MeanReducerState reduce(MeanReducerState acc, Object input) {
        acc.count++;
        if(input instanceof Double) {
            acc.sum += (Double)input;
        } else if(input instanceof Long) {
            acc.sum += ((Long)input).doubleValue();
        } else if(input instanceof Integer) {
            acc.sum += ((Integer)input).doubleValue();
        } else {
            throw new RuntimeException(
                "MeanReducer::reduce called with unsupported input type `" + input.getClass()
                + "`. Supported types are Double, Long, Integer.");
        }
        return acc;
    }

    public Object extractResult(MeanReducerState acc) {
        if(acc.count > 0) {
            return new Double(acc.sum / (double)acc.count);
        } else {
            return null;
        }
    }
}
```

14.2　内置统计类型

Spout与Bolt对应着相似的统计类型，但二者略有区别。例如，Spout和Bolt都含有消息被Ack的数目，但只有Bolt含有执行数目统计信息（execute-count）。本节将分别对两种类型的内置统计进行讨论。而运行统计的更新时机与上一章讨论的内容相同，本章不再赘述。

14.2.1 Spout类型的内置统计

表14-1列出了与Spout相对应的统计类型及其更新方法。

表14-1 Spout的内置统计类型

名　称	类　型	更新方法	描　述
ack-count	MultiCountMetric	spout-acked-tuple!	Ack数目
complete-latency	MultiReducedMetric	spout-acked-tuple!	处理时延
fail-count	MultiCountMetric	spout-failed-tuple!	失败数目
emit-count	MultiCountMetric	emitted-tuple!	发送数目
transfer-count	MultiCountMetric	transferred-tuple!	传输数目

emit-count指的是一个组件发送的消息数。但在某些情况下，一条消息要被复制多份再发送出去，这时transfer-count则可表示实际的消息传输数目。例如，子节点以全局分组的方式接收某一个流，那么该流中的一条消息将被发送到所有子节点上，传输数目为接收子节点的数目，即同一条消息被发送多次。

14.2.2 Bolt类型的内置统计

表14-2列举出了与Bolt相对应的统计类型及其更新方法。

表14-2 Bolt的内置统计类型

名　称	类　型	更新方法	描　述
ack-count	MultiCountMetric	bolt-acked-tuple!	Ack数目
process-latency	MultiReducedMetric	bolt-acked-tuple!	处理时延
fail-count	MultiCountMetric	bolt-failed-tuple!	失败数目
execute-count	MultiCountMetric	bolt-execute-tuple!	执行数目
execute-latency	MultiReducedMetric	bolt-execute-tuple!	执行时延
emit-count	MultiCountMetric	emitted-tuple!	发送数目
transfer-count	MultiCountMetric	transferred-tuple!	传输数目

处理时延（process-latency）表示为从收到该消息到对该消息进行Ack操作之间的时间，而执行时延（execute-latency）指的是调用Bolt的execute方法的执行时间。

14.3 统计触发消息

在创建Executor时，Storm会调用setup-ticks!方法为该Executor设置用于发送触发统计消息的计时器，其作用为以固定的时间间隔向特定的流发送消息。Executor在收到该消息后，就会将运行统计信息发送出去然后将其清空。触发统计消息并不会真正地从Executor中发送出去，且只

有Executor自身可以接收到它，其作用在于提示Executor定期执行一些Task。相关代码如下：

```
(setup-ticks! worker executor-data)
```

14.3.1　注册统计信息

本节讨论Storm是如何创建这些统计信息并将它们注册到系统中的。

1. 内置统计的注册方法

TopologyContext对象含有如下两个成员变量：

```
private Map<Integer,Map<Integer, Map<String, IMetric>>> _registeredMetrics;
private clojure.lang.Atom _openOrPrepareWasCalled;
```

- _registeredMetrics用来存储所有的注册统计类别。它的键为统计时间间隔，由参数TOPOLOGY_BUILTIN_METRICS_BUCKET_SIZE_SECS设定。其值为另外一个哈希表，键为TaskId，该哈希表的值也为一个哈希表，其键为GlobalStreamId，值为对应的IMetrics。这个IMetrics用来保存在某一个统计时间间隔里某个Task上某个流的统计信息。_registeredMetrics变量对应于Executor的数据interval->task->metric-registry，且由Executor创建，故一个Executor内部的所有Task将共享该变量。
- _openOrPrepareWasCalled变量表示 Bolt的prepare或者Spout的open方法是否已经被调用。registerMetric方法只能在prepare或者open方法之中或之前调用。

TopologyContext对象的registerMetric方法的代码如下，其中形参name为统计的名字，timeBucketSizeInSecs表示统计的间隔：

```
public <T extends IMetric> T registerMetric(String name, T metric, int timeBucketSizeInSecs) {
    if((Boolean)_openOrPrepareWasCalled.deref() == true) {
        throw new RuntimeException("TopologyContext.registerMetric can only be called from within
            overridden " +
                                   "IBolt::prepare() or ISpout::open() method.");
    }

    Map m1 = _registeredMetrics;
    if(!m1.containsKey(timeBucketSizeInSecs)) {
        m1.put(timeBucketSizeInSecs, new HashMap());
    }

    Map m2 = (Map)m1.get(timeBucketSizeInSecs);
    if(!m2.containsKey(_taskId)) {
        m2.put(_taskId, new HashMap());
    }

    Map m3 = (Map)m2.get(_taskId);
    if(m3.containsKey(name)) {
        throw new RuntimeException("The same metric name `" + name + "` was registered twice." );
    } else {
        m3.put(name, metric);
    }
```

```
    return metric;
}
```

2. 创建并注册统计信息

在与Task对应的mk-task-data函数中，会调用make-data函数来创建内置的统计信息，这些统计数据对应于Task数据中的builtin-metrics变量。

这里，make-data函数用于根据传入的节点类型来创建与Spout或Bolt相对应的运行统计类别对象，其代码如下：

```
(defn make-data [executor-type]
    (condp = executor-type
        :spout (BuiltinSpoutMetrics. (MultiCountMetric.)
                   (MultiReducedMetric. (MeanReducer.))
                   (MultiCountMetric.)
                   (MultiCountMetric.)
                   (MultiCountMetric.))
        :bolt (BuiltinBoltMetrics. (MultiCountMetric.)
                  (MultiReducedMetric. (MeanReducer.))
                  (MultiCountMetric.)
                  (MultiCountMetric.)
                  (MultiReducedMetric. (MeanReducer.))
                  (MultiCountMetric.)
                  (MultiCountMetric.))))
```

之后，Executor会在创建消息循环线程时，将这些统计信息注册到TopologyContext对象中，相关代码如下：

```
(builtin-metrics/register-all (:builtin-metrics task-data) storm-conf (:user-context task-data))
```

register-all方法的形参:builtin-metrics为通过make-data函数创建的统计集合，其代码如下：

```
(defn register-all [builtin-metrics  storm-conf topology-context]
    (doseq [[kw imetric] builtin-metrics]
        (.registerMetric topology-context (str "__" (name kw)) imetric
            (int (get storm-conf Config/TOPOLOGY_BUILTIN_METRICS_BUCKET_SIZE_SECS)))))
```

可以看出，内置统计的名字是以“__”作为前缀的。目前，Storm中只有一种统计的时间间隔设置项TOPOLOGY_BUILTIN_METRICS_BUCKET_SIZE_SECS，其默认值为60秒。

14.3.2 触发消息的产生与发送

setup-metrics!函数用来设置计时器，该计时器将定期地向__tick流发送消息。当Task收到这些消息后，统计信息就会被发送出去，于是系统中的统计节点便可以收到这些统计信息。该函数的实现如下：

```
1 (defn setup-metrics! [executor-data]
2     (let [{:keys [storm-conf receive-queue worker-context interval->task->metric-registry]}
          executor-data
```

```
3             distinct-time-bucket-intervals (keys interval->task->metric-registry)]
4        (doseq [interval distinct-time-bucket-intervals]
5            (schedule-recurring
6                (:user-timer (:worker executor-data))
7                interval
8                interval
9                (fn []
10                   (disruptor/publish
11                       receive-queue
12                         [[nil (TupleImpl. worker-context [interval] Constants/SYSTEM_TASK_ID
                             Constants/METRICS_TICK_STREAM_ID)]]))))))
```

- 第3行获得统计间隔的个数，目前只有一个，间隔的大小为60秒。
- 第5行设置一个计时器，其时间间隔为统计的时间间隔。
- 第9~12行为定时器的回调函数的实现。该函数将向Executor的接收消息队列发送一条消息，这个消息的TaskId为`nil`，表明该Executor所包含的Task都要执行该消息。消息的内容为统计间隔，该消息会被发送至流`__tick`。

14.3.3　处理统计触发消息

当Executor收到来自`__tick`流的消息后，将调用`metrics-tick`函数来准备和发送统计消息，其中统计消息中含有该节点的运行统计结果。`metrics-tick`函数的实现如下：

```
1 (defn metrics-tick [executor-data task-datas ^TupleImpl tuple]
2     (let [{:keys [interval->task->metric-regiretry ^WorkerTopologyContext worker-context]}
         executor-data
3             interval (.getInteger tuple 0)]
4     (doseq [[task-id task-data] task-datas
5               :let [name->imetric (-> interval->task->metric-registry (get interval) (get
                  task-id))
6                   task-info (IMetricsConsumer$TaskInfo.
7                                 (. (java.net.InetAddress/getLocalHost) getCanonicalHostName)
8                                 (.getThisWorkerPort worker-context)
9                                 (:component-id executor-data)
10                                task-id
11                                (long (/ (System/currentTimeMillis) 1000))
12                                interval)
13                   data-points (->> name->imetric
14                                   (map (fn [[name imetric]]
15                                       (let [value (.getValueAndReset ^IMetric imetric)]
16                                           (if value
17                                              (IMetricsConsumer$DataPoint. name value)))))
18                                 (filter identity)
19                                 (into []))]]
20      (if (seq data-points)
21         (task/send-unanchored task-data Constants/METRICS_STREAM_ID [task-info data-points])))))
```

- 在Executor中，关键字`interval->task->metric-regiretry`存储着目前的运行统计，第3行获取消息的内容，即将获得的统计时间间隔存储到`interval`变量。

- 第4行遍历Task的数据task-datas，Executor所包含的每一个Task都将被进行这一处理。
- 第5行中name->imetric为该统计间隔interval以及该task-id对应的统计信息。
- 第6~12行用于构建task-info对象。该对象包含该Task的主机名、Worker的端口号、组件、taskId、当前时间以及统计间隔。
- 第13~19行用于构建data-points。这里会对name->imetric中的每一条记录调用getValue-AndReset函数，以获得记录的当前值并清空，然后过滤掉其中的空类型元素并将结果存储到数组中。
- 在第20~21行中，若data-points不为空，则发送消息[task-info，data-points]到流METRICS_STREAM_ID中。

当一个Executor中存在多个Task时，该方法是存在一定问题的。由于处理一条统计触发消息时，Executor中所有的Task（而非特定的Task）都会执行一次，并且每次执行过程中，所有的Task均会被再次执行，因此第一次执行时将所有Task的统计值设置为0，再次执行时，则会将统计值0作为消息内容发送出去。例如，Executor中含有两个Task，收到一条统计触发消息时，metrics-tick函数将被执行两次。然而，每次执行时，第4行又对每一个Task进行执行，故将产生问题。

14.4 运行统计收集节点

Storm会根据配置文件中TOPOLOGY-METRICS-CONSUMER-REGISTER的设置来创建运行统计的收集节点。metrics-consumer-bolt-specs函数用来创建需要的这些节点，相关代码如下：

```
1 (defn metrics-consumer-bolt-specs [components-ids-that-emit-metrics storm-conf]
2    (let [inputs (->> (for [comp-id components-ids-that-emit-metrics]
3                          {[comp-id METRICS-STREAM-ID] :shuffle})
4                      (into {}))
5
6             mk-bolt-spec (fn [class arg p]
7                            (thrift/mk-bolt-spec*
8                                inputs
9                                (backtype.storm.metric.MetricsConsumerBolt. class arg)
10                               {} :p p :conf {TOPOLOGY-TASKS p}))]
11
12     (map
13         (fn [component-id register]
14             [component-id (mk-bolt-spec (get register "class")
15                               (get register "argument")
16                               (or (get register "parallelism.hint") 1))])
17
18     (metrics-consumer-register-ids storm-conf)
19     (get storm-conf TOPOLOGY-METRICS-CONSUMER-REGISTER))))
```

- 第2~4行获得所有组件的METRICS-STREAM-ID流，并设置为Shuffle分组方式，然后将它们存储到变量inputs中。这为创建统计收集节点做准备，任何一个统计收集节点都需要收听所有节点的METRICS-STREAM-ID流。

14

- 第6~10行定义了一个函数mk-bolt-spec，它利用thrift/mk-bolt-spec*函数创建了一个Bolt节点。这里inputs为输入流集合，class和arg为构建Bolt所需要依据的参数，节点类型为MetricsConsumerBolt。设置并行度为p，默认值为0，表示该节点并没有放入Storm系统中，这也是内置的运行统计没有在Storm中开启的原因之一。
- 第13~16行的函数将遍历第18行和第19行的集合，其中mk-bolt-spec函数用来定义一个组件。例如，系统中注册了两个相同的Metrics类MyMetricsConsumer，则第一个的名字为__metrics_org.mycompany.MyMetricsConsumer,第二个的名字为__metrics_org.mycompany.MyMetricsConsumer#2。感兴趣的读者可以阅读metrics-consumer-register-ids、number-duplicates以及map-occurrences函数，学习该名字是如何计算得到的。
- 第18行利用metrics-consumer-register-ids函数为注册的统计收集节点命名，该函数返回一个集合。
- 第19行获得注册的统计节点集合。

下面简要介绍这些系统节点是如何嵌入在Storm的Topology中的。

add-metric-components!函数会调用metrics-consumer-bolt-specs函数，并将得到的Bolt添加到Topology中，其代码如下：

```
(defn add-metric-components! [storm-conf ^StormTopology topology]
    (doseq [[comp-id bolt-spec] (metrics-consumer-bolt-specs (keys (all-components topology))
        storm-conf)]
      (.put_to_bolts topology comp-id bolt-spec)))
```

在生成Worker对应数据的过程中，当初始化:task->component关键字所对应的值时，将调用storm-task-info函数未对该变量赋值，变量：task->component中含有从taskId到组件的映射关系。

storm-task-info函数将在用户定义的Topology的基础上调用system-topology!函数，为Topology添加一些系统Bolt以协助系统工作。例如，本章所讨论的统计收集节点就是通过add-metric-components!函数来添加的。storm-task-info函数的代码如下：

```
(defn storm-task-info
    "Returns map from task -> component id"
    [^StormTopology user-topology storm-conf]
    (->> (system-topology! storm-conf user-topology)
         all-components
         (map-val (comp #(get % TOPOLOGY-TASKS) component-conf))
         (sort-by first)
         (mapcat (fn [[c num-tasks]] (repeat num-tasks c)))
         (map (fn [id comp] [id comp]) (iterate (comp int inc) (int 1)))
         (into {})
         ))
```

sytem-topology!函数用于添加系统节点以向系统提供支持，该函数的定义如下，其中add-metric-components!函数用来添加统计信息收集节点：

```
(defn system-topology! [storm-conf ^StormTopology topology]
    (validate-basic! topology)
```

```
    (let [ret (.deepCopy topology)]
        (add-acker! storm-conf ret)
        (add-metric-components! storm-conf ret)
        (add-metric-streams! ret)
        (add-system-streams! ret)
        (add-system-components! ret)
        (validate-structure! ret)
        ret
        ))
```

add-metric-streams! 函数用于将统计流作为系统中每一个节点的输出流。于是，用户定义的Spout和Bolt节点就不需要声明该流，而却可以向该流发送消息了，其输出模式为["task-info" "data-points"]。该函数的实现如下：

```
(defn add-metric-streams! [^StormTopology topology]
    (doseq [[_ component] (all-components topology)
            :let [common (.get_common component)]]
        (.put_to_streams common METRICS-STREAM-ID
            (thrift/output-fields ["task-info" "data-points"]))))
```

MetricsConsumerBolt将接受这些运行统计信息。MetricsConsumerBolt类将接收一个实现了IMetricsConsumer的类作为参数，在prepare函数中生成该类的一个对象，在execute函数中调用该类的handleDataPoints函数，该函数的输入为TaskInfo以及DataPoints。相关代码如下：

```
@Override
public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {
    try {
        _metricsConsumer = (IMetricsConsumer)Class.forName(_consumerClassName).newInstance();
    } catch (Exception e) {
        throw new RuntimeException("Could not instantiate a class listed in config under section" +
            Config.TOPOLOGY_METRICS_CONSUMER_REGISTER + " with fully qualified name " +
                _consumerClassName, e);
    }
    _metricsConsumer.prepare(stormConf, _registrationArgument, context, (IErrorReporter)
        collector);
    _collector = collector;
}

@Override
public void execute(Tuple input) {
    _metricsConsumer.handleDataPoints((IMetricsConsumer.TaskInfo)input.getValue(0), (Collection)
        input.getValue(1));
        _collector.ack(input);
    }
```

14.5 SystemBolt

SystemBolt是系统中的一种特殊Bolt，每个Worker都会启动一个这样的Bolt。它的TaskId为-1，因此不会有用户消息发送至该Bolt，但它会接收统计信息触发消息。

该Bolt会注册一些自定义的统计信息，并在收到流METRICS_TICK_STREAM_ID中的消息后，

调用该Task所对应的metrics-tick方法。于是，这些统计信息便被发送了出去并且重置，而达到了Worker所在进程的统计信息则会被发送到运行统计收集节点上去。该Bolt注册的统计信息如表14-3所示。

表14-3 SystemBolt中含有的运行统计

统计信息	描 述
MemoryUsageMetric	获得该进程的内存使用情况，主要有如下统计： ❑ maxBytes ❑ committed ❑ initBytes ❑ usedBytes ❑ virtualFreeBytes ❑ unusedBytes 在具体实现中，该统计通过调用MemoryMXBean的方法获得
GarbageCollectorMetric	进程垃圾回收情况，含有如下统计： ❑ Count ❑ timeMs 在具体实现中，该统计通过调用GarbageCollectorMXBean的方法获得
uptimeSecs	启动时间。 在具体实现中，该统计通过调用RuntimeMXBean.getUptime()方法获得
startTimeSecs	RuntimeMXBean.getStartTime()
newWorkerEvent	首次创建Worker时为1
memory/heap	RuntimeMXBean.getHeapMemoryUsage
memory/nonHeap	RuntimeMXBean.getNonHeapMemoryUsage

读者可以参考MemoryUsageMetric和GarbageCollectorMetric的类实现来学习如何实现用户自定义统计。

类似地，add-system-components!函数可将SystemBolt添加到Worker中，它可以被system-topology!函数调用，相关代码如下：

```
(defn add-system-components! [conf ^StormTopology topology]
    (let [system-bolt-spec (thrift/mk-bolt-spec*
            {}
            (SystemBolt.)
            {SYSTEM-TICK-STREAM-ID (thrift/output-fields ["rate_secs"])
                METRICS-TICK-STREAM-ID (thrift/output-fields ["interval"])}
            :p 0
            :conf {TOPOLOGY-TASKS 0})]
        (.put_to_bolts topology SYSTEM-COMPONENT-ID system-bolt-spec)))
```

可以看出，该Bolt只接收SYSTEM-TICK-STREAM以及METRICS-TICK-STREAM。

第 15 章 事务Topology的实现

事务类型的Topology是Storm中的关键实现之一，在处理消息的过程中，它将保证消息不会被丢失。如果再结合具有去重功能的存储，就可以实现消息被整个系统处理一次且仅处理一次。

Storm项目的作者Nathan在下面的链接中提到了一些有关事务Topology的关键概念：https://github.com/nathanmarz/storm/wiki/Transactional-topologies。徐明明翻译了这篇文章并给出了一些个人见解，相关链接为http://xumingming.sinaapp.com /736/twitter-storm-transactional-topolgoy/。

本章将对事务类型的Topology的实现进行系统的分析。

15.1 事务 Topology 的实现概述

事务Topology的实现概述如下。

- 事务类型的Spout节点实际上是一个子Topology，它包含一个协调Spout节点（Coordinator）以及一些消息发送Bolt节点（Emitter）。
- 协调Spout节点的并行度为1，消息发送Bolt节点的并行度则可根据需要来设定。
- 协调Spout节点并不发送实际的数据，而是将事务尝试发送到消息将Bolt节点中。事务尝试（TransactionalAttempt）包含一个BigInteger类型的事务号以及长整型类型的尝试号。当事务被重传时，事务号是相同的，但是尝试号不同。
- 协调Spout节点与消息发送Bolt节点之间采用全局分组方式进行消息传输，也就意味着从协调Spout中发送的一条事务尝试消息都会被所有的消息发送Bolt节点接收。每个消息发送Bolt节点会根据收到的事务尝试消息来发送与该事务对应的消息集合。消息发送Bolt节点之间则需要对该事务所对应的数据进行协作，即每个节点只负责事务的一部分数据。
- 当所有的消息发送节点都成功处理了该事务在该节点上所对应的消息后（通过Storm的Ack框架），协调Spout节点认为该事务已经被成功处理，协调Spout节点将会产生并发送下一个事务尝试消息。
- 协调Spout节点中含有两个系统输出流：事务消息流（Batch流）和事务提交流（Commit流）协调Spout节点会向事务消息流发送事务尝试消息，向事务提交流发送事务的提交消息。
- 事务的处理实际上被分成以下两个阶段。
    - 事务处理阶段：协调Spout节点向消息发送节点发送事务尝试消息，消息发送Bolt节点

发送该事务所对应的消息集合，然后Storm系统开始处理这些消息。该阶段属于事务的处理阶段。

- 事务提交阶段：通过系统的Ack框架，当系统中的消息均被成功处理后，协调Spout节点将收到其发送的事务尝试消息的Ack，这表明事务处理阶段已经结束。

在事务类型的Topology中，可以定义事务提交Bolt（Commit Bolt），这种类型的Bolt节点可保证其处理的事务会按照顺序被提交。事务的提交操作可能是将事务的处理结果存储起来。

在Storm中，若消息丢失或者超时，一个事务可能会被重做，此时将导致消息重复。但由于提交节点可以保证事务是按照顺序提交的，此处更利于去重操作。例如，若每个事务对应的数据在重传时相同，则在提交节点看到相同的事务序号时，可认为该事务是重传的事务，进而将其忽略掉。

当事务的处理阶段完成后，协调Spout节点会检查该事务是否为下一个要提交的事务，若是则将该事务的事务序号发送到事务提交流。所有的事务提交Bolt节点都会接收该流的消息，并会在收到事务提交消息后，对该事务进行提交。可以看出，当事务处理阶段结束后，并不一定会立即进入事务提交阶段，它需要等待之前的事务都已经被成功提交后，方可进入事务提交阶段。此处，保证了事务按顺序提交。

通过系统的Ack框架，在收到事务提交消息的Ack之后，该事务被认为已成功处理。

在事务提交节点，将调用节点的`finishBatch`方法完成事务的提交。其他的处理节点（BatchBolt）同样含有`finishBatch`回调方法，但它却表示在该节点上所有属于同一事务的消息都已经被处理。本章将在`CoordinatedBolt`类的实现中详细讨论Storm是如何保证`finishBatch`方法被正确调用的。

在事务处理阶段，属于一个事务的消息以及所有衍生出来的消息均以协调Spout节点发送的事务尝试消息为根。基于Storm的Ack框架，当所有的消息均被成功处理之后，协调Spout节点将收到一条事务尝试消息的Ack。若消息处理失败或者超时，协调Spout节点则会收到失败消息，事务类型的Topology此时会对该事务进行重传处理。这样，事务类型的Topology便可以保证消息不丢失。实际上，事务尝试消息会存储于ZooKeeper中，所以即便Topology异常停止运行，仍可保证在其重新启动时事务能被正确重传。

根据其中Spout类型的不同，事务Topology可被进一步分为基本事务Topology和模糊事务Topology（Opaque Transactional Topology），接下来详细讨论一下。

15.1.1　事务Topology的类型

根据Topology中Spout类型的不同，Topology可以分为如下两种类型。

- 非事务Topology（Non-Transactional）：Spout的类型为`IRichSpout`。Storm并不保证消息的可靠传输，故消息可能会丢失。
- 事务Topology：Spout的类型为`ITransactionalSpout`。此时，Storm负责初始化一个事务，并负责在事务失败时对其进行重传。事务Topology保证了消息的可靠传输，以及在事务提交节点处，事务按顺序被提交。根据对`ITransactionalSpout`接口的不同实现，事务Topology可以进一步分为两种类型。
  - 基本事务Topology：每个事务所对应的数据在事务被重传时不发生变化。用户只要保证

数据的元数据不变，就可每次获取到相同的数据集合。例如，Spout的数据源为一个大的数据文件，每个事务负责读取数据文件的一部分。这时，可以将事务的元数据设计为某部分数据在数据文件中的位移量及数量。于是，当事务被重传时，根据该元数据，事务所对应的数据不会发生变化，这就满足了基本事务Topology所需的前提条件。

- 模糊事务Topology：每个事务所对应的数据在事务重传时可能发生变化，但会保证事务中的消息只属于某一个事务，即保证同一条消息不会同时属于多个事务。例如，在Spout从Kafka队列读取数据时，事务的元数据可以为Kafka队列中的位移量；当事务重传时，Kafka队列中可能已经含有了新的数据，于是被重传的事务可以将这些新消息包含在其中。此时，重传事务与原始事务所对应的消息不同，但每条消息却只属于同一个事务，故满足了模糊事务Topology的前提条件。

不同事务类型的Topology对应于不同的去重办法。对于基本事务Topology，结果集合只需要存储事务序号以及相应的事务处理结果即可。若收到相同的事务序号时，只需要将数据丢弃；而在收到不同的事务序号时，才对数据进行更新。对于模糊事务Topology，仅仅保存当前的事务序号并不足够，因为即便事务序号相同，其对应的数据也可能不同，此时还需要保存该事务在更新前所对应的数据。在第19章中，我们会对事务数据的去重进行较为详细的介绍。

此外，根据`ITransactionalSpout`的具体实现，又可以将事务类型的Topology分为分区的事务Topology以及非分区的事务Topology。对于分区的事务Topology，每个发送节点只负责特定的数据分区，它更适用于读取Kafka 队列。

15.1.2　事务Topology的类关系

事务 Topology 的类关系如图 15-1 所示。

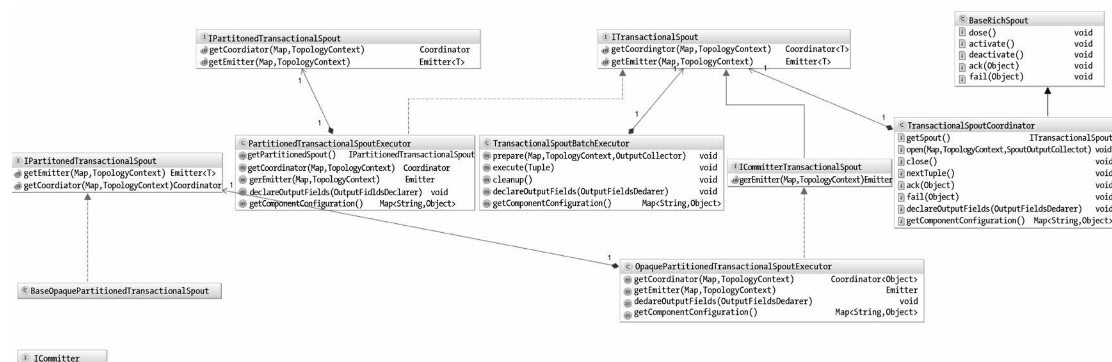


图15-1　事务Topology的类关系

通过表15-1，我们可以初步了解这些类和接口。学习这些类的实现是理解事务类型的Topology的关键。

表15-1　事务Topology的接口以及实现类

接口或者类	描　　述
ITransactionalSpout	基本事务Topology的Spout接口，内含两部分接口：协调Spout接口以及消息发送Bolt接口
TransactionalSpoutBatchExecutor	Bolt类型，用于执行ITransactionalSpout中的消息发送Bolt节点
TransactionalSpoutCoordinator	Spout类型，用于执行ITransactionalSpout中的协调Spout节点，是系统中唯一的Spout节点，具体功能为初始化事务以及产生事务尝试消息等
IPartitionedTransactionalSpout	分区的事务Topology的Spout接口，用户通过该接口来完成Spout的分区功能
PartitionedTransactionalSpoutExecutor	为ITransactionalSpout类型，主要用于适配IPartitionedTransactionalSpout接口，为IPartitionedTransactionalSpout的执行器
IOpaquePartitionedTransactionalSpout	分区的模糊事务Topology的Spout接口，用户通过该接口来完成模糊事务类型的Topology
OpaquePartitionedTransactionalSpoutExecutor	为ITransactionalSpout类型，用于适配IOpaquePartitionedTransactionalSpout接口，为IOpaquePartitionedTransactionalSpout的执行器
ICommitterTransactionalSpout	具有提交功能的事务Topology的Spout接口，主要用于模糊事务Topology的Spout节点

15.2　ITransactionalSpout 接口

ITransactionalSpout接口中包含两部分接口，一部分接口用于提供协调Spout节点的逻辑，一部分接口用于提供消息发送Bolt节点的逻辑。用户只需要实现这些接口，Storm便会负责将这些逻辑部署到合适的节点上运行。仔细阅读接口的注释对于了解如何使用该接口很有帮助。该接口的定义如下：

```
public interface ITransactionalSpout<T> extends IComponent {
    public interface Coordinator<X> {
        /**
        * Create metadata for this particular transaction id which has never
        * been emitted before. The metadata should contain whatever is necessary
        * to be able to replay the exact batch for the transaction at a later point.
        *
        * The metadata is stored in Zookeeper.
        *
        * Storm uses the Kryo serializations configured in the component configuration
        * for this spout to serialize and deserialize the metadata.
        *
        * @param txid The id of the transaction.
        * @param prevMetadata The metadata of the previous transaction
        * @return the metadata for this new transaction
        */
        X initializeTransaction(BigInteger txid, X prevMetadata);

        /**
        * Returns true if its ok to emit start a new transaction, false otherwise (will skip this
        * transaction).
```

```
        * You should sleep here if you want a delay between asking for the next transaction (this will
        * be called repeatedly in a loop).
        */
        boolean isReady();

        /**
        * Release any resources from this coordinator.
        */
        void close();
    }

    public interface Emitter<X> {
        /**
        * Emit a batch for the specified transaction attempt and metadata for the transaction. The
        * metadata was created by the Coordinator in the initializeTranaction method. This method must
          always emit the same batch of tuples across all tasks for the same transaction id.
        *
        * The first field of all emitted tuples must contain the provided TransactionAttempt.
        *
        */
        void emitBatch(TransactionAttempt tx, X coordinatorMeta, BatchOutputCollector collector);

        /**
        * Any state for transactions prior to the provided transaction id can be safely cleaned up,
        * so this method should clean up that state.
        */
        void cleanupBefore(BigInteger txid);

        /**
        * Release any resources held by this emitter.
        */
        void close();
    }

    /**
    * The coordinator for a TransactionalSpout runs in a single thread and indicates when batches
    * of tuples should be emitted and when transactions should commit. The Coordinator that you provide
    * in a TransactionalSpout provides metadata for each transaction so that the transactions can be
    * replayed.
    */
    Coordinator<T> getCoordinator(Map conf, TopologyContext context);
    /**
    * The emitter for a TransactionalSpout runs as many tasks across the cluster. Emitters are responsible
    * for emitting batches of tuples for a transaction and must ensure that the same batch of tuplesis
    * always emitted for the same transaction id.
    */
    Emitter<T> getEmitter(Map conf, TopologyContext context);
}
```

协调Spout节点的isReady方法用来检测当前是否可以开始一个新事务。在Storm中，许多地方都可能是产生新事务的合适时间点。例如，在Spout收到Ack消息后，上一个事务可能已经处理结束，此时可以调用isReady方法来判断是否可以开始一个新事务。类似地，在Spout节点的nextTuple方法中，也会调用isReady方法来判断是否可以开始一个新事务。从前面对Executor的讨论可以知

道，Spout的主循环线程会依次处理输入消息以及产生新的消息，这要求其中调用的方法应是非阻塞的。因此，isReady方法需要设计为非阻塞的，也即如果新事务并不就绪，就立即返回false，而不应该调用Sleep方法。注释中建议使用睡眠方法是不正确的。

initializeTransaction方法用于产生新事务的元数据，它可以基于上一个事务的元数据来产生。在初始化第一个事务时，prevMetadata为空。注意，该方法仅当isReady方法返回true时才有可能被调用到，并且对于每个事务它只会调用一次。

在消息发送Bolt节点的接口中，emitBatch方法最为重要。协调Spout节点发送的事务尝试消息都会到达消息发送Bolt节点，然后该节点会调用emitBatch方法来发送一批数据。这个过程要保证同一个事务序号对应于相同的数据（模糊事务类型的Spout除外）。注意，对于同一个事务序号，该方法可能会被调用多次（譬如事务被重传时）。

cleanupBefore方法只有在与输入的事务序号相对应的事务全被成功处理后才会被调用，它负责给用户提供合适的时间点来清理与事务相关的数据。

15.3　协调 Spout 节点的执行器

与ITransactionalSpout中的协调Spout接口相对应的逻辑需要放在一个Spout节点中来执行，该Spout节点是系统中唯一真正的Spout节点，它产生事务尝试消息并将消息发送给消息发送Bolt节点。

TransactionalSpoutCoordinator类负责产生并维护这些事务尝试消息，它会调用协调Spout接口中用户定义的逻辑，是事务Topology中非常重要的部分。理解这个类对于理解事务 Topology非常关键。

为了实现消息的可靠重传，所有正在处理的事务尝试消息都会被存储在ZooKeeper里，Storm提供了两个工具类用于维护这些消息。本章首先讨论事务Topology对ZooKeeper的使用。

15.3.1　ZooKeeper客户端工具

在事务Topology中，TransactionalState类用于存储事务的元数据，它是基于Curator Framework实现的。Curator Framework是Apache下面的一个开源软件，它为更好地使用ZooKeeper提供了支持，详情可访问http://curator.incubator.apache.org/curator-framework/。

TransactionalState类主要用于维护ZooKeeper中某一个目录下面的所有子节点。在事务Topology中，每个正在处理的事务都对应该目录下面的一个节点。TransactionalState类的实现如下：

```
1    public class TransactionalState {
2        CuratorFramework _curator;
3        KryoValuesSerializer _ser;
4        KryoValuesDeserializer _des;
5
6        public static TransactionalState newUserState(Map conf, String id, Map componentConf) {
7            return new TransactionalState(conf, id, componentConf, "user");
8        }
9
```

```
10      public static TransactionalState newCoordinatorState(Map conf, String id, Map componentConf)
          {
11          return new TransactionalState(conf, id, componentConf, "coordinator");
12        }
13
14        protected TransactionalState(Map conf, String id, Map componentConf, String subroot) {
15            try {
16                conf = new HashMap(conf);
17                // ensure that the serialization registrations are consistent with the
                  declarations in this spout
18                if(componentConf!=null) {
19                    conf.put(Config.TOPOLOGY_KRYO_REGISTER,
20                                componentConf
21                                .get(Config.TOPOLOGY_KRYO_REGISTER));
22                  }
23                String rootDir = conf.get(Config.TRANSACTIONAL_ZOOKEEPER_ROOT) + "/" + id + "/"
                      + subroot;
24              List<String> servers = (List<String>) getWithBackup(conf,
                  Config.TRANSACTIONAL_ZOOKEEPER_SERVERS, Config.STORM_ZOOKEEPER_SERVERS);
25          Object port = getWithBackup(conf, Config.TRANSACTIONAL_ZOOKEEPER_PORT, Config.STORM_
              ZOOKEEPER_PORT);
26            CuratorFramework initter = Utils.newCuratorStarted(conf, servers, port);
27            try {
28                initter.create().creatingParentsIfNeeded().forPath(rootDir);
29            } catch(KeeperException.NodeExistsException e)  {
30
31            }
32
33            initter.close();
34
35            _curator = Utils.newCuratorStarted(conf, servers, port, rootDir);
36            _ser = new KryoValuesSerializer(conf);
37            _des = new KryoValuesDeserializer(conf);
38        } catch (Exception e) {
39           throw new RuntimeException(e);
40        }
41    }
42
43    public void setData(String path, Object obj) {
44        path = "/" + path;
45        byte[] ser = _ser.serializeObject(obj);
46        try {
47            if(_curator.checkExists().forPath(path)!=null) {
48                _curator.setData().forPath(path, ser);
49            } else {
50                _curator.create()
51                        .creatingParentsIfNeeded()
52                        .withMode(CreateMode.PERSISTENT)
53                        .forPath(path, ser);
54            }
55        } catch(Exception e) {
56            throw new RuntimeException(e);
57        }
58    }
```

```
59
60
61
62      public List<String> list(String path) {
63          path = "/" + path;
64          try {
65              if(_curator.checkExists().forPath(path)==null) {
66                  return new ArrayList<String>();
67              } else {
68                  return _curator.getChildren().forPath(path);
69              }
70          } catch(Exception e) {
71              throw new RuntimeException(e);
72          }
73      }
74
75      public void mkdir(String path) {
76          setData(path, 7);
77      }
78
79      public Object getData(String path) {
80          path = "/" + path;
81          try {
82              if(_curator.checkExists().forPath(path)!=null) {
83                  return _des.deserializeObject(_curator.getData().forPath(path));
84              } else {
85                  return null;
86              }
87          } catch(Exception e) {
88              throw new RuntimeException(e);
89          }
90      }
91
92      public void close() {
93          _curator.close();
94      }
95
96      private Object getWithBackup(Map amap, Object primary, Object backup) {
97          Object ret = amap.get(primary);
98          if(ret==null) return amap.get(backup);
99          return ret;
100     }
101 }
```

- 第2行定义一个Curator的成员变量。
- 第3~4行定义序列化以及反序列化的成员变量，这里利用Kyro进行序列化。对于要存放到ZooKeeper中的自定义类，需要进行Kyro的注册，否则将无法写入ZooKeeper。
- 第6~12行定义了两个工具方法，分别对应于user和coordinator子目录。其中coordinator子目录用于为协调 Spout节点存储元数据。关于“user”子目录，本书会在15.6.2节进一步讨论。
- 第17~22行用来获取用户注册的使用Kyro序列化的类型。

- 第23行产生事务Topology用于存储元数据的ZooKeeper路径。通常,TRANSACTIONAL_ZOOKEEPER_ROOT的值为/transactional；id为Spout节点名字，该名字是在创建Topology时指定的；subroot为user或者coordinator。例如，若Spout的名字为TestSpout，则元数据的路径为/transactional/TestSpout/coordinator。每次提交同样的Topology时，Storm会产生不同的TopologyId以作区分，但Spout节点名字是固定的。于是重新提交的Topology会继续访问上一个Topology在ZooKeeper中存储的元数据，也就实现了从上一个Topology结束点继续执行的目标。这也是事务Topology可以实现可靠消息传输的原因之一。
- 第24~25行用来获取ZooKeeper的服务器名称及端口号。Storm本身需要利用ZooKeeper来存储一些元数据，而事务Topology中存储的元数据更接近于用户数据，原则上应使用分开的ZooKeeper服务器进行存储。Storm提供了这样的灵活性，如果设置了TRANSACTIONAL_ZOOKEEPER_SERVERS，则Storm会利用该配置项制定的ZooKeeper来存储数据，否则就使用默认的STORM_ZOOKEEPER_SERVERS。
- 第26~31行利用CuratorFramework来创建根目录。由于根目录是相对固定的，只需要创建一次，因此ZooKeeper在创建节点时并不会再次创建根目录。例如，若目录结构为/a/b/c/d，则在创建节点d之前需要预先依次为其创建父节点a、b、c。
- 第35行重新创建Curator对象并传入已经创建好的根目录。接下来的ZooKeeper操作均基于这个根目录，相比于每次都重新根据ZooKeeper的根目录“/”来确定存储路径的过程，Storm采用的这种方法提高了使用ZooKeeper的效率。
- 第36~37行创建序列化和反序列化对象。
- 第43~58行是setData方法的实现。由于ZooKeeper中只能存储字节数组，因此该方法会首先将输入的对象序列化。如果数据节点已经存在，则直接设置数据；若不存在，则创建节后再设置数据。注意，传入的路径path可以继续含有子目录结构。
- 第62~73行将某一个目录结构下的全部子节点返回。当重新提交的Topology开始运行后，就可以通过该方法获取那些需要被重新执行的事务。其他的函数比较类似，这里不再一一讨论。
- 第96~100行的getWithBackup函数优先使用primary定义的值，如果不存在，则使用backup对应的值。

RotatingTransactionalState类是对TransactionalState的进一步封装,它提供了额外的两个功能：首先，通过在内存中维护一个TreeMap结构以优化查询速度；其次，将不需要的数据（如已经完成的事务的元数据）从ZooKeeper中删除。ZooKeeper中存储的数据节点增多可能会导致各种各样的问题，于是清理ZooKeeper中不再使用的数据是非常关键且必需的步骤。该类的代码如下：

```
1    /**
2    * A map from txid to a value. Automatically deletes txids that have been committed.
3    */
4    public class RotatingTransactionalState {
5        public static interface StateInitializer {
6            Object init(BigInteger txid, Object lastState);
```

```
7       }
8
9       private TransactionalState _state;
10      private String _subdir;
11      private boolean _strictOrder;
12
13      private TreeMap<BigInteger, Object> _curr = new TreeMap<BigInteger, Object>();
14
15      public RotatingTransactionalState(TransactionalState state, String subdir, Boolean
            strictOrder) {
16          _state = state;
17          _subdir = subdir;
18          _strictOrder = strictOrder;
19          state.mkdir(subdir);
20          sync();
21       }
22
23       public RotatingTransactionalState(TransactionalState state, String subdir) {
24           this(state, subdir, false);
25       }
26
27       public Object getLastState() {
28           if(_curr.isEmpty()) return null;
29           else return _curr.lastEntry().getValue();
30       }
31
32       public void overrideState(BigInteger txid, Object state) {
33           _state.setData(txPath(txid), state);
34           _curr.put(txid, state);
35       }
36
37       public void removeState(BigInteger txid) {
38           if(_curr.containsKey(txid)) {
39               _curr.remove(txid);
40               _state.delete(txPath(txid));
41           }
42       }
43
44       public Object getState(BigInteger txid, StateInitializer init) {
45           if(!_curr.containsKey(txid)) {
46               SortedMap<BigInteger, Object> prevMap = _curr.headMap(txid);
47               SortedMap<BigInteger, Object> afterMap = _curr.tailMap(txid);
48
49               BigInteger prev = null;
50               if(!prevMap.isEmpty()) prev = prevMap.lastKey();
51
52               if(_strictOrder) {
53                   if(prev==null && !txid.equals(TransactionalSpoutCoordinator.INIT_TXID)) {
54                     throw new IllegalStateException("Trying to initialize transaction for which
                           there should be a previous state");
55                   }
56                   if(prev!=null && !prev.equals(txid.subtract(BigInteger.ONE))) {
57                       throw new IllegalStateException("Expecting previous txid state to be the
                           previous transaction");
```

```
58                  }
59                  if(!afterMap.isEmpty()) {
60                      throw new IllegalStateException("Expecting tx state to be initialized in
                            strict order but there are txids after that have state");
61                  }
62              }
63
64
65              Object data;
66              if(afterMap.isEmpty()) {
67                  Object prevData;
68                  if(prev!=null) {
69                      prevData = _curr.get(prev);
70                  } else {
71                      prevData = null;
72                  }
73                  data = init.init(txid, prevData);
74              } else {
75                  data = null;
76              }
77              _curr.put(txid, data);
78              _state.setData(txPath(txid), data);
79          }
80          return _curr.get(txid);
81      }
82
83      public boolean hasCache(BigInteger txid) {
84          return _curr.containsKey(txid);
85      }
86
87      /**
88       * Returns null if it was created, the value otherwise.
89       */
90      public Object getStateOrCreate(BigInteger txid, StateInitializer init) {
91          if(_curr.containsKey(txid)) {
92              return _curr.get(txid);
93          } else {
94              getState(txid, init);
95              return null;
96          }
97      }
98
99      public void cleanupBefore(BigInteger txid) {
100         SortedMap<BigInteger, Object> toDelete = _curr.headMap(txid);
101         for(BigInteger tx: new HashSet<BigInteger>(toDelete.keySet())) {
102             _curr.remove(tx);
103             _state.delete(txPath(tx));
104         }
105     }
106
107     private void sync() {
108         List<String> txids = _state.list(_subdir);
109         for(String txid_s: txids) {
110             Object data = _state.getData(txPath(txid_s));
```

```
111                 _curr.put(new BigInteger(txid_s), data);
112             }
113         }
114 
115         private String txPath(BigInteger tx) {
116             return txPath(tx.toString());
117         }
118 
119         private String txPath(String tx) {
120             return _subdir + "/" + tx;
121         }
122 }
```

- 第5~7行定义接口StateInitializer，该接口用于表明如何初始化事务元数据。在事务Topology中，该接口中定义的init方法将调用ITransactionalSpout. initializeTransaction方法来产生新事务所对应的元数据。
- 第9~11行定义了3个成员变量，其中_state对象用来操作ZooKeeper，_subdir表示子目录，_strictOrder则用来表明创建的事务是否需要满足强序列。由于Storm可以根据上一个事务的元数据来初始化当前事务，所以事务之间是存在一定联系的，这时就需要保证事务初始化的强序列。基础事务Topology中使用了强序列。
- 第13行定义了_curr成员变量，它为TreeMap对象。其存储的数据与ZooKeeper中的一致，即Storm同时维护了内存中的数据及ZooKeeper中的数据。
- 第15~21行定义的构造函数用于对象的初始化。第20行会调用sync方法，根据ZooKeeper中已经存在的数据构建_curr对象。sync方法在第107~113行定义。可以看出，Storm用事务序号作为ZooKeeper路径的一部分。
- 第27~42行定义了一些读写方法。读时会从_curr返回结果；写时首先更新ZooKeeper中的数据，然后更新内存中的数据。
- 第44~81行定义了getState方法，其参数为BigInteger类型的事务序号，以及一个用于初始化一个新事务的回调函数init。如果该事务序号所对应的元数据已经被创建，则在第80行直接返回；若未创建，这个类首先将_curr分成两个部分，一部分是均比事务序号要小的事务，将其放在preMap中，另一部分是均比事务序号要大的事务，将其放在afterMap中。prev表示比当前事务序号小的最大的事务序号。如果_strictOrder为true，将进行如下检查。
  - prev为空，并且当前事务序号不为1。
  - prev不为空，并且prev!=txid-1。
  - afterMap为空，即含有比当前txid更大序号的事务已经初始化了。
- 第65~76行计算当前事务所对应的元数据。第66~73行对应于当前事务序号即为最大事务序号的情况，此时Storm会获取上一个事务的元数据，同时调用init方法获取当前事务的元数据。第75行对应于当前的事务序号并不是最大事务序号的情况，这种情况也即当_strictOrder为false的情况，这表明已经有后面的事务被初始化了，并且利用的是当前事务的前一个事务的元数据，此时Storm会将当前事务的元数据设置为空。在非强序的情

况下，用户代码应更加小心，它需要根据当前获得的元数据是否为空来进行之后的操作，或者将其忽略或者给予特殊地处理。

- 第77~78行依次更新内存及ZooKeeper中的数据。个人认为颠倒这两行的顺序可能更好，即先更新ZooKeeper中的数据。
- 第90~97行定义了`getStateOrCreate`方法。如果元数据已经创建过，则返回该元数据；如果未创建，则调用`getState`方法创建，但返回值为空。它主要用在消息发送Bolt节点上，以表明是第一次处理事务。这个方法的目的是有些奇怪的。
- 第99~105行定义的`cleanupBefore`函数负责对已经完成的事务进行清理。由于事务提交是强序的，所以当前事务被提交时，之前的事务都已经完成了，故清理这些数据是安全的。利用TreeMap结构来获得需要清除的事务序号是非常方便的。读者可以思考一下，如果直接操作ZooKeeper，又应如何实现。

在下一节中，我们将讨论如何基于`TransactionalState`和`RotatingTransactionalState`类来实现协调Spout的执行器。

15.3.2　协调Spout的执行器

`TransactionalSpoutCoordinator`类继承自类`BaseRichSpout`，而`BaseRichSpout`又将实现接口`IRichSpout`，具体的类关系如图15-2所示。

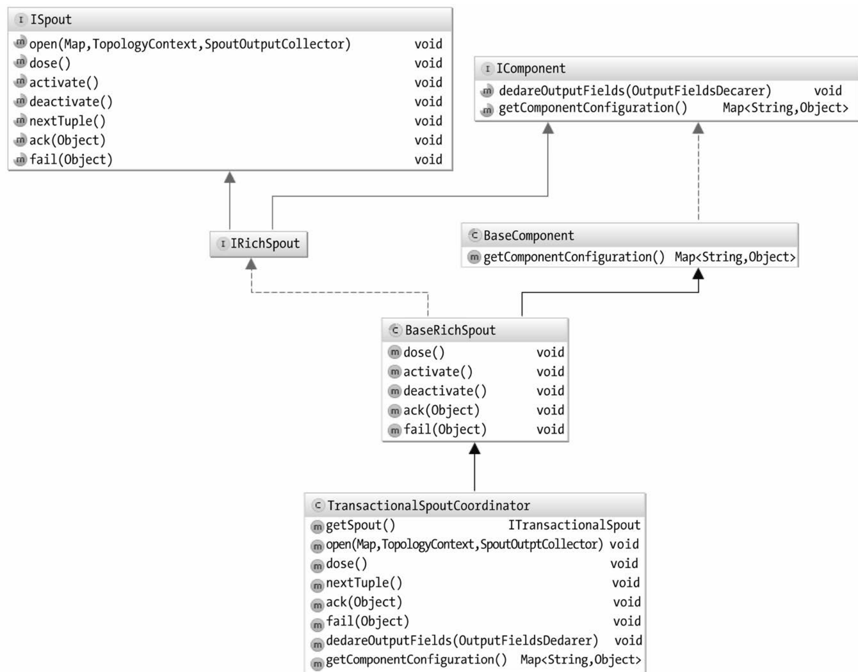


图 15-2　`TransactionalSpoutCoordinator` 的类关系

TransactionalSpoutCoordinator类使用内部私有类AttemptStatus来表示一个事务尝试的3种状态，其代码如下：

```
private static enum AttemptStatus {
    PROCESSING,
    PROCESSED,
    COMMITTING
}
```

其中各个成员变量的含义如下。

- PROCESSING：事务尝试消息已经从协调Spout发送出去，表明当前事务所对应的数据已经开始处理。
- PROCESSED：该事务已经被处理完成并等待提交。
- COMMITTING：该事务之前的事务都已被提交，于是发送消息到提交Bolt，表示当前的事务也可以被提交了。在协调Spout收到这条消息的Ack后，当前事务就被认为是已经成功处理了，此时可以开始一个新事务。

接下来，分析 TransactionalSpoutCoordinator 类的实现，其代码如下：

```
1     public class TransactionalSpoutCoordinator extends BaseRichSpout {
2     public static final Logger LOG = LoggerFactory.getLogger(TransactionalSpoutCoordinator.class);
3
4         public static final BigInteger INIT_TXID = BigInteger.ONE;
5
6         public static final String TRANSACTION_BATCH_STREAM_ID = TransactionalSpoutCoordinator.
              class.getName() + "/batch";
7         public static final String TRANSACTION_COMMIT_STREAM_ID = TransactionalSpoutCoordinator.
              class.getName() + "/commit";
8
9         private static final String CURRENT_TX = "currtx";
10        private static final String META_DIR = "meta";
11
12        private ITransactionalSpout _spout;
13        private ITransactionalSpout.Coordinator _coordinator;
14        private TransactionalState _state;
15        private RotatingTransactionalState _coordinatorState;
16
17        TreeMap<BigInteger, TransactionStatus> _activeTx = new TreeMap<BigInteger, Transaction
              Status>();
18
19        private SpoutOutputCollector _collector;
20        private Random _rand;
21        BigInteger _currTransaction;
22        int _maxTransactionActive;
23        StateInitializer _initializer;
24
25
26        public TransactionalSpoutCoordinator(ITransactionalSpout spout) {
27            _spout = spout;
28        }
```

```
29
30      public ITransactionalSpout getSpout() {
31          return _spout;
32      }
33
34      @Override
35      public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
36          _rand = new Random(Utils.secureRandomLong());
37          _state = TransactionalState.newCoordinatorState(conf, (String)
                conf.get(Config.TOPOLOGY_TRANSACTIONAL_ID), _spout.getComponentConfiguration());
38          _coordinatorState = new RotatingTransactionalState(_state, META_DIR, true);
39          _collector = collector;
40          _coordinator = _spout.getCoordinator(conf, context);
41          _currTransaction = getStoredCurrTransaction(_state);
42          Object active = conf.get(Config.TOPOLOGY_MAX_SPOUT_PENDING);
43          if(active==null) {
44              _maxTransactionActive = 1;
45          } else {
46              _maxTransactionActive = Utils.getInt(active);
47          }
48          _initializer = new StateInitializer();
49      }
50
51      @Override
52      public void close() {
53          _state.close();
54      }
55
56      @Override
57      public void nextTuple() {
58          sync();
59      }
60
61      @Override
62      public void declareOutputFields(OutputFieldsDeclarer declarer) {
63        // in partitioned example, in case an emitter task receives a later transaction than it's
64         // emitted so far, when it sees the earlier txid it should know to emit nothing
65        declarer.declareStream(TRANSACTION_BATCH_STREAM_ID, new Fields("tx", "tx-meta",
          "committed-txid"));
66        declarer.declareStream(TRANSACTION_COMMIT_STREAM_ID, new Fields("tx"));
67    }
68
69      @Override
70      public Map<String, Object> getComponentConfiguration() {
71          Config ret = new Config();
72          ret.setMaxTaskParallelism(1);
73          return ret;
74      }
75
76      private BigInteger nextTransactionId(BigInteger id) {
77          return id.add(BigInteger.ONE);
78      }
79
80      private BigInteger previousTransactionId(BigInteger id) {
```

```
81          if(id.equals(INIT_TXID)) {
82              return null;
83          } else {
84              return id.subtract(BigInteger.ONE);
85          }
86      }
87
88      private BigInteger getStoredCurrTransaction(TransactionalState state) {
89          BigInteger ret = (BigInteger) state.getData(CURRENT_TX);
90          if(ret==null) return INIT_TXID;
91          else return ret;
92      }
93 }
```

- 第4行将INIT_TXID定义为1，表示事务序号都是从1开始的。
- 第6行定义TRANSACTION_BATCH_STREAM_ID，事务尝试消息将被发送到这个流。所有消息发送Bolt都会通过直接分组的方式来接收这个流的消息。它的模式为<"tx","tx-meta","committed-txid">，其中tx为TransactionalAttempt对象，tx-meta为用户自定义的事务元数据，committed-txid表示上一个已经提交的事务序号（在Bolt中，可以利用该值对数据进行清理）。
- 第7行定义TRANSACTION_COMMIT_STREAM_ID，用于发送事务提交的消息。所有的事务提交Bolt节点都会通过直接分组的方式来接收这个流的消息。它的模式是<"tx">，作用为通知事务提交Bolt事务txid已经处理结束了。txId对应的事务及其之前的事务都已经处理结束是一个非常重要的时间点，该时间点保证了finishBatch是按照事务的顺序被调用的。
- 第9~10行定义了ZooKeeper中两个子目录的名称，currtx用来存储当前等待提交的事务序号，meta用来存储系统中所有正在执行的事务序号。
- 第12~13行定义代理的ITransactionalSpout及其协调Spout对象。
- 第14~15行定义的对象用于操作ZooKeeper，_state用来维护当前的事务，_coordinatorState则用来维护正在处理的事务。
- 第17行的_activeTx表示正在处理事务。与_coordinateState不同，_activeTx还保存着当前事务处于何种状态的信息。这里涉及的TransactionStatus类包含attempt和status这两个对象，其代码如下：

```
private static class TransactionStatus {
    TransactionAttempt attempt;
    AttemptStatus status;

    public TransactionStatus(TransactionAttempt attempt) {
        this.attempt = attempt;
        this.status = AttemptStatus.PROCESSING;
    }

    @Override
    public String toString() {
        return attempt.toString() + " <" + status.toString() + ">";
```

```
        }
    }
```

- 第20行定义的_rand随机函数用于产生事务的尝试序号。事务在进行重传时，其事务序号是相同的，而事务尝试序号是不同的。
- 第23行实现接口StateInitializer，它会调用initializeTransaction方法来初始化一个与事务相对应的元数据,其代码如下：

```
private class StateInitializer implements RotatingTransactionalState.StateInitializer {
    @Override
    public Object init(BigInteger txid, Object lastState) {
        return _coordinator.initializeTransaction(txid, lastState);
    }
}
```

- 第35~49行实现了Spout的open方法。该方法将负责与ZooKeeper中的元数据同步，将之前未处理完的事务的元数据恢复到内存中，并进行重传。这是事务Topology可以实现可靠处理的基础。在第38行中，初始化RotatingTransactionalState对象的最后一个参数被设置为true，表示将开启对元数据对象初始化顺序的检查。
- 第42行中，参数TOPOLOGY_MAX_SPOUT_PENDING表示最多可以有多少个事务在系统中被同时执行，其默认值为1。

在分析TransactionalSpoutCoordinator的其他方法之前，我们首先来介绍sync方法，其代码如下：

```
1 private void sync() {
2         // note that sometimes the tuples active may be less than max_spout_pending, e.g.
3         // max_spout_pending = 3
4         // tx 1, 2, 3 active, tx 2 is acked. there won't be a commit for tx 2 (because tx 1 isn't
             committed yet),
5         // and there won't be a batch for tx 4 because there's max_spout_pending tx active
6         TransactionStatus maybeCommit = _activeTx.get(_currTransaction);
7         if(maybeCommit!=null && maybeCommit.status == AttemptStatus.PROCESSED) {
8             maybeCommit.status = AttemptStatus.COMMITTING;
9             _collector.emit(TRANSACTION_COMMIT_STREAM_ID, new Values(maybeCommit.attempt),
                 maybeCommit.attempt);
10        }
11
12        try {
13            if(_activeTx.size() < _maxTransactionActive) {
14                BigInteger curr = _currTransaction;
15                for(int i=0; i<_maxTransactionActive; i++) {
16                    if((_coordinatorState.hasCache(curr) || _coordinator.isReady())
17                         && !_activeTx.containsKey(curr)) {
18                        TransactionAttempt attempt = new TransactionAttempt(curr, _rand.
                             nextLong());
19                        Object state = _coordinatorState.getState(curr, _initializer);
20                        _activeTx.put(curr, new TransactionStatus(attempt));
21                        _collector.emit(TRANSACTION_BATCH_STREAM_ID, new Values(attempt, state,
```

```
                              previousTransactionId(_currTransaction)), attempt);
22                   }
23                   curr = nextTransactionId(curr);
24               }
25           }
26      } catch(FailedException e) {
27          LOG.warn("Failed to get metadata for a transaction", e);
28      }
29  }
```

- 第2~5行的注释是很重要的，即表示目前活跃的事务序号为1、2、3，此时事务序号2已经进行Ack操作了，但是由于事务序号1还没有进行Ack操作，所以当前的事务序号currTxId还只能为1。于是，此时系统中活跃的事务只有事务1和3，少于参数maxSpoutPending所定义的活跃事务数目。
- 第6~10行检查_currTransaction所对应的事务是否已经处于PROCESSED状态，如果是，则将事务的状态改成COMMITTING，并向提交消息流发送消息，同时将TransactionalAttempt作为消息跟踪ID，Storm需要对这条消息进行跟踪。只有在收到该消息的Ack之后，_currTransaction才会指向下一个事务序号，这是保证提交消息为强序的关键。
- 第12~25行对可能的事务进行初始化。若_activeTx的元素数目比_maxTransactionActive的小，表示可以继续初始化新的事务并使其运行，此时Storm采用了一种更为激进的方式，它会逐一检查从事务序号_currTransaction到事务序号_currTransaction+_maxTransactionActive-1的事务是否都已经被初始化。这可以在某种程度上抑制数据不同步的情况产生。
- 第16~17行的条件很重要，可以这么理解。
  - _activeTx中不包含当前事务curr，但_coordinatorState中却包含了当前事务curr。这表明_coordinatorState与_activeTx不同步，系统将重新发送这条事务消息。这是有可能发生的，例如一个Topology在被杀掉后重新提交，_coordinatorState中会包含上一个Topology未完成的事务序号，而_activeTx中却并不会包含这些信息，Storm需要对这样的事务进行重传处理。
  - _activeTx中不包含当前事务curr，_coordinatorState中也不包含当前事务curr，同时isReady函数返回为true，此时表明可以开始一个新事务，并且数据已经准备好了，系统此时将发送一条新的事务尝试消息。这部分逻辑还是有问题的，例如若maxTransactionActive=2，当txId=1时，isReady返回为false，序号为1的事务并未初始化成功；然而当txId=2时，isReady返回为true，Storm试图初始化序号为2的事务。但是由于序号为1的事务并未初始化，此时将不满足元数据的强序检查， Storm会因此推出异常。Topology在刚刚启动时更容易发生这个问题。

接下来，对Spout的ack方法进行分析，其代码如下：

```
1 @Override
2 public void ack(Object msgId) {
```

```
3     TransactionAttempt tx = (TransactionAttempt) msgId;
4     TransactionStatus status = _activeTx.get(tx.getTransactionId());
5     if(status!=null && tx.equals(status.attempt)) {
6         if(status.status==AttemptStatus.PROCESSING) {
7             status.status = AttemptStatus.PROCESSED;
8         } else if(status.status==AttemptStatus.COMMITTING) {
9             _activeTx.remove(tx.getTransactionId());
10            _coordinatorState.cleanupBefore(tx.getTransactionId());
11            _currTransaction = nextTransactionId(tx.getTransactionId());
12            _state.setData(CURRENT_TX, _currTransaction);
13        }
14        sync();
15    }
16 }
```

- ack方法可能收到从两个发送流输出的消息的Ack，但是这里没有办法区分，只能根据事务的状态进行区分。msgId实际上为TransactionAttempt对象。
- 在第5行中，如果Ack回来的事务尝试消息与_activeTx中的不同，则忽略这条消息，它很可能是某个已经丢弃的重传的事务。
- 在第6~7行中，如果状态为PROCESSING，表明数据处理已经结束了，因此随即将其状态改成PROCESSED。
- 在第8~13行中，若状态为COMMITTING，表明收到了从事务提交Bolt返回来的Ack消息，意味着事务处理已经结束，接下来进行数据清理。第12行将_currTransaction更新为下一个事务序号，以保证强序关系。
- 第14行调用了sync方法，因为此时有机会产生新事务。

最后对Spout的fail方法进行分析，其实现较为简单，相关代码如下：

```
@Override
public void fail(Object msgId) {
    TransactionAttempt tx = (TransactionAttempt) msgId;
    TransactionStatus stored = _activeTx.remove(tx.getTransactionId());
    if(stored!=null && tx.equals(stored.attempt)) {
        _activeTx.tailMap(tx.getTransactionId()).clear();
        sync();
    }
}
```

注意当某一个事务处理失败时，Storm将重传这个事务之后的所有事务。

TransactionalSpoutCoordinator类还有一些其他方法，但都比较直观，这里不再赘述。

15.3.3 消息发送Bolt的执行器

类似地，TransactionalSpoutBatchExecutor类用来执行ITransactionalSpout中消息发送Bolt的接口，它是Bolt类型的。

我们首先介绍接口ICommitterTransactionalSpout，其代码如下：

```
public interface ICommitterTransactionalSpout<X> extends ITransactionalSpout<X> {
    public interface Emitter extends ITransactionalSpout.Emitter {
        void commit(TransactionAttempt attempt);
    }

    @Override
    public Emitter getEmitter(Map conf, TopologyContext context);
}
```

该接口为ITransactionalSpout.Emitter添加了一个commit方法。通常，事务的元数据是在协调Spout中产生并被清理的，但它也可能由消息发送节点在emitBatch时产生，此时协调Spout节点并没有足够的数据对元数据进行操作，因此元数据需要在消息发送节点中进行处理。目前，这个接口主要用来支持模糊事务Topology。

下面分析TransactionalSpoutBatchExecutor的实现，该类实现IRichBolt接口，表示该节点本质上为Bolt节点，其代码如下：

```
1    public class TransactionalSpoutBatchExecutor implements IRichBolt {
2        public static Logger LOG = LoggerFactory.getLogger(TransactionalSpoutBatchExecutor.class);
3
4        BatchOutputCollectorImpl _collector;
5        ITransactionalSpout _spout;
6        ITransactionalSpout.Emitter _emitter;
7
8        TreeMap<BigInteger, TransactionAttempt> _activeTransactions = new TreeMap<BigInteger,
             TransactionAttempt>();
9
10       public TransactionalSpoutBatchExecutor(ITransactionalSpout spout) {
11           _spout = spout;
12       }
13
14       @Override
15       public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
16           _collector = new BatchOutputCollectorImpl(collector);
17           _emitter = _spout.getEmitter(conf, context);
18       }
19
20       @Override
21       public void cleanup() {
22           _emitter.close();
23       }
24
25       @Override
26       public void declareOutputFields(OutputFieldsDeclarer declarer) {
27           _spout.declareOutputFields(declarer);
28       }
29
30       @Override
31       public Map<String, Object> getComponentConfiguration() {
32           return _spout.getComponentConfiguration();
33       }
34 }
```

- 第5~6行定义了类的成员变量_spout和_emitter，其中后者由_spout.getEmitter方法获得，即用户定义的消息发送节点。
- 第8行定义的_activeTransactions维护了正在运行的事务。不同于协作Spout节点，消息发送节点可能存在多个并行度。_activeTransactions主要为模糊事务Topology服务。

该类的其他方法比较简单，这里主要分析其execute方法的实现，其代码如下：

```
1 @Override
2 public void execute(Tuple input) {
3     TransactionAttempt attempt = (TransactionAttempt) input.getValue(0);
4     try {
5          if(input.getSourceStreamId().equals(TransactionalSpoutCoordinator.TRANSACTION_COMMIT
              _STREAM_ID)) {
6             if(attempt.equals(_activeTransactions.get(attempt.getTransactionId()))) {
7                 ((ICommitterTransactionalSpout.Emitter) _emitter).commit(attempt);
8                 _activeTransactions.remove(attempt.getTransactionId());
9                 _collector.ack(input);
10            } else {
11                _collector.fail(input);
12            }
13        } else {
14            _emitter.emitBatch(attempt, input.getValue(1), _collector);
15            _activeTransactions.put(attempt.getTransactionId(), attempt);
16            _collector.ack(input);
17            BigInteger committed = (BigInteger) input.getValue(2);
18            if(committed!=null) {
19                // valid to delete before what's been committed since
20                // those batches will never be accessed again
21                _activeTransactions.headMap(committed).clear();
22                _emitter.cleanupBefore(committed);
23            }
24        }
25     } catch(FailedException e) {
26         LOG.warn("Failed to emit batch for transaction", e);
27         _collector.fail(input);
28     }
29 }
```

- 第3行从输入消息的第1列获得事务尝试消息。在事务Topology中，限定第1列的值必须为事务尝试消息。
- 在第6~12行中，若输入的消息来自于流TRANSACTION_COMMIT_STREAM_ID，则表示当前的事务已经结束执行，可以开始对元数据的提交操作。只有当_spout实现了接口ICommitterTransactionalSpout后，Spout节点才会接收协调Spout节点的TRANSACTION_COMMIT_STREAM_ID流。由于消息发送节点的并行度可以大于1，所以对于同一个事务，元数据提交方法commit可能会被调用多次。本书会在模糊事务Topology的实现中对其进行进一步分析。
- 第14~23行处理来自于TRANSACTION_BATCH_STREAM_ID流的消息。输入消息的第1列为事务所对应的元数据。第14行调用_emitter的emitBatch方法。第15行对输入的消息进行Ack操作。输入消息的第2列为已经提交的事务序号，表示该事务之前的事务都已经被成功处理了，

于是可以据此清理变量_activeTransactions。

- 第22行调用_emitter的cleanupBefore回调方法，以清理用户代码中的历史事务数据。

15.4　CoordinatedBolt 的实现分析

CoordinatedBolt是非常关键的一个类，它用于协调系统中的Bolt节点。对于事务Topology在一个Bolt节点中，若能获知属于某一个事务的消息是否已经全部收到，这将是非常有帮助的。

CoordinatedBolt会记录自身发送至目标节点的消息数目，当属于一个事务的消息发送结束后，它通过直接分组的方式向协调流发送一条消息，通知目标节点需要从该CoordinatedBolt接收多少条消息。

事务Topology中Bolt节点都利用CoordinatedBolt进行包装，它期待从上游节点收到这些协调消息以判断属于一个事务的消息是否收全，并在收全一个事务的消息后，向它的下游节点发出其需要接收的消息数目的通知。

CoordinatedBolt节点收到并处理完属于一个事务的全部消息后，将调用finishBatch方法。finishBatch方法是事务Topology中非常重要的方法，在事务提交节点中，它用于存储事务处理的结果。当它被调用时，就表示属于一个事务的消息都已经被成功处理了。

15.4.1　TrackingInfo

TrackingInfo是CoordinatedBolt将记录的那些信息。每个事务尝试消息都会对应一个TrackingInfo对象，也即哈希表_tracked成员变量的键为事务尝试消息。重传的事务将会对应不同的TrackingInfo对象。理解这个类里面的成员，对于理解Coordinated Bolt是非常重要的。该类的代码如下：

```
private TimeCacheMap<Object, TrackingInfo> _tracked;
public static class TrackingInfo {
    int reportCount = 0;
    int expectedTupleCount = 0;
    int receivedTuples = 0;
    boolean failed = false;
    Map<Integer, Integer> taskEmittedTuples = new HashMap<Integer, Integer>();
    boolean receivedId = false;
    boolean finished = false;
    List<Tuple> ackTuples = new ArrayList<Tuple>();
}
```

下面简要介绍该类的成员变量。

- reportCount：用来表示已经收到的数据源的个数。一个Bolt节点可能从多个节点接收消息，每个上游节点都需要向其发送协调消息。
- expectedTupleCount：表示期望收到消息的总数目。这个数目会根据收到的协调消息进行更新，该协调消息来自协调消息流。

- receivedTuples：表示已经收到消息的总数目，它会在CoordinatedOutputCollector中更新。
- failed：用来表示该事务尝试是否已经失败。
- taskEmittedTuples：用来表示该Bolt发送出去的消息数目，以目标节点号作为键。当该Bolt已经成功处理完一个事务之后，它会向每一个目标节点发送一条协调消息，通知目标节点需要从它这里收到多少条消息。该协调消息将被发送到协调消息流。
- receivedId：如果Bolt实现了ICommitter接口，那么只有在收到从协调Spout节点发送过来的事务提交消息后，该变量才会被设置为true。对于普通的Bolt，则默认就被设为true，即普通Bolt节点中处理的事务序号不是强序的。
- finished：用来表示该事务处理是否已经结束。
- ackTuples：用来表示收到的元数据消息。该消息有两种类型，一种是从协调Spout收到的事务提交消息，另外一种是从其他协调Bolt收到的协调消息。在事务Topology中，Storm只对控制消息进行跟踪，而对普通的数据消息则不通过Ack框架进行跟踪。

15.4.2 CoordinatedOutputCollector

CoordinatedBolt使用CoordinatedOutputCollector发送数据。CoordinatedOutputCollector类实现了IOutputCollector接口，它采用装饰模式，含有一个IOutputCollector的代理，并且添加了消息统计功能。它会统计发送到目标节点的消息数目，并在事务处理完成后根据这些统计到的消息数目构建协调消息，然后将它们发送至相应的目标节点。该类的代码如下：

```
1  public class CoordinatedOutputCollector implements IOutputCollector {
2      IOutputCollector _delegate;
3
4      public CoordinatedOutputCollector(IOutputCollector delegate) {
5          _delegate = delegate;
6      }
7
8      public List<Integer> emit(String stream, Collection<Tuple> anchors, List<Object> tuple) {
9          List<Integer> tasks = _delegate.emit(stream, anchors, tuple);
10         updateTaskCounts(tuple.get(0), tasks);
11         return tasks;
12     }
13
14     public void emitDirect(int task, String stream, Collection<Tuple> anchors, List<Object>
           tuple) {
15         updateTaskCounts(tuple.get(0), Arrays.asList(task));
16         _delegate.emitDirect(task, stream, anchors, tuple);
17     }
18
19     public void ack(Tuple tuple) {
20         Object id = tuple.getValue(0);
21         synchronized(_tracked) {
22             TrackingInfo track = _tracked.get(id);
23             if (track != null)
24                 track.receivedTuples++;
```

```
25          }
26          boolean failed = checkFinishId(tuple, TupleType.REGULAR);
27          if(failed) {
28              _delegate.fail(tuple);
29          } else {
30              _delegate.ack(tuple);
31          }
32      }
33
34      public void fail(Tuple tuple) {
35          Object id = tuple.getValue(0);
36          synchronized(_tracked) {
37              TrackingInfo track = _tracked.get(id);
38              if (track != null)
39                  track.failed = true;
40          }
41          checkFinishId(tuple, TupleType.REGULAR);
42          _delegate.fail(tuple);
43      }
44
45      public void reportError(Throwable error) {
46          _delegate.reportError(error);
47      }
48
49
50      private void updateTaskCounts(Object id, List<Integer> tasks) {
51          synchronized(_tracked) {
52              TrackingInfo track = _tracked.get(id);
53              if (track != null) {
54                  Map<Integer, Integer> taskEmittedTuples = track.taskEmittedTuples;
55                  for(Integer task: tasks) {
56                      int newCount = get(taskEmittedTuples, task, 0) + 1;
57                      taskEmittedTuples.put(task, newCount);
58                  }
59              }
60          }
61      }
62  }
```

- 第2~6行定义了构造函数，需要传入所代理的输出收集器。Storm中采用了较多的装饰模式，最基本的输出收集器均是在executor.clj中定义的，然后因为可以在其基础上进行定制并添加功能。
- 第8~12行定义了emit方法。由于该类是CoordinatedBolt的一个内部类，所以它可以访问CoordinatedBolt中的类成员变量。_tracked是CoordinatedBolt定义的成员变量，在事务Topology中，用于跟踪每一个事务尝试在这个Bolt上的处理情况。该方法除了进行正常的消息发送外，还会更新发送到每一个目标节点的消息数目的。
- 第19~32行定义ack方法，并且统计已经收到的消息数目，其中会调用checkFinishId方法检测当前事务是否已经失败，然后调用代理类的Ack或者Fail方法。通常，CoordinatedBolt会被BatchBoltExecutor类包装，后者的execute方法会对输入的消息进行Ack操作，于是

本类中的ack方法将被调用，故可在此处统计收到的消息数目。

- 第34~43行定义了fail方法，它将TrackingInfo数据标记为失败。
- 第50~61行实现了更新已经发送的消息数目的方法。

在Storm的Ack框架下，最终的Fail和Ack消息会传送到Spout节点，而Spout节点会随即调用其Ack或者Fail方法，然后用户的逻辑便知道了是否该重传或者产生下一条消息。中间的Bolt节点并不会重传任何消息。这也简化了Storm的消息管理。

15.4.3 CoordinatedBolt中的消息类型

CoordinatedBolt主要处理的3种消息类型如下。

- REGULAR：正常的数据消息。
- ID：从协调Spout节点收到的事务提交消息。
- COORD：其他的CoordinatedBolt收到的协调消息。

CoordinatedBolt会根据输入消息的流号来对消息的类型进行判断。Topology构建器会将实现了ICommitter的Bolt中的_idStreamSpec设为协调Spout节点的事务提交流。其他情况下，_idStreamSpec的值为空。如果消息来自协调消息流，则认为消息的类型为协调消息类型COORD，其他情况下默认为REGULAR类型。

目前，Storm还不存在一种机制能禁止用户向系统的流中写入数据。显而易见，如果写入的话，将会产生很多莫名其妙的错误。

消息的类型定义及判定方法如下面的代码所示：

```
static enum TupleType {
    REGULAR,
    ID,
    COORD
}

private TupleType getTupleType(Tuple tuple) {
    if(_idStreamSpec!=null
        && tuple.getSourceGlobalStreamid().equals(_idStreamSpec._id)) {
        return TupleType.ID;
    } else if(!_sourceArgs.isEmpty()
        && tuple.getSourceStreamId().equals(Constants.COORDINATED_STREAM_ID)) {
        return TupleType.COORD;
    } else {
        return TupleType.REGULAR;
    }
}
```

15.4.4 成员变量以及主要方法分析

CoordinatedBolt成员变量的定义如下：

```
private Map<String, SourceArgs> _sourceArgs;
private IdStreamSpec _idStreamSpec;
private IRichBolt _delegate;
private Integer _numSourceReports;
private List<Integer> _countOutTasks = new ArrayList<Integer>();;
private OutputCollector _collector;
private TimeCacheMap<Object, TrackingInfo> _tracked;
```

- _sourceArgs：用来表示哪些节点将向该Bolt发送协调消息。
- _idStreamSpec：目前用来表示该节点是否是事务提交节点。
- _delegate：内含实际的Bolt逻辑。
- _numSourceReports：表示Bolt的上游节点的个数。
- _countOutTasks：表示将向哪些Task发送数据。
- _collector：可以统计消息发送和接收数目的输出收集器。
- _tracked：用来保存在此节点中正在被处理的事务尝试，键为事务尝试消息，值为该事务的处理情况。它是TrackingInfo类的对象。

接下来分析一下主要的实现方法。首先，我们看看execute方法的实现：

```
1  public void execute(Tuple tuple) {
2      Object id = tuple.getValue(0);
3      TrackingInfo track;
4      TupleType type = getTupleType(tuple);
5      synchronized(_tracked) {
6          track = _tracked.get(id);
7          if(track==null) {
8              track = new TrackingInfo();
9              if(_idStreamSpec==null) track.receivedId = true;
10             _tracked.put(id, track);
11         }
12     }
13
14     if(type==TupleType.ID) {
15         synchronized(_tracked) {
16             track.receivedId = true;
17         }
18         checkFinishId(tuple, type);
19     } else if(type==TupleType.COORD) {
20         int count = (Integer) tuple.getValue(1);
21         synchronized(_tracked) {
22             track.reportCount++;
23             track.expectedTupleCount+=count;
24         }
25         checkFinishId(tuple, type);
26     } else {
27         synchronized(_tracked) {
28             _delegate.execute(tuple);
29         }
30     }
31 }
```

- ❑ 第4行计算获得输入消息的消息类型。
- ❑ 第5~12行从第1列得到消息的事务尝试消息，然后将其作为键，若该事务还没有被跟踪，则开始对其进行跟踪。
- ❑ 在第9行中，如果_idStreamSpec为空，则直接将receivedId设置为true，即在非事务提交节点中会这样设置。
- ❑ 第14~18行表示收到了从协调Spout发送过来的事务提交消息。
- ❑ 第19~25行表示收到了从其他节点发送过来的协调信息，此时Storm将更新跟踪信息中的reportCount以及expectedTupleCount。协调消息的模式为<Id, Count>。
- ❑ 第26~29行处理普通的数据消息。

在处理控制消息时，会调用checkFinishId方法来检测该节点是否完成了对事务的处理，进而决定是否可以调用finishBatch方法以及是否可以向其下游节点发送协调消息等。checkFinishId方法的分析如下，该方法是CoordinatedBolt的核心，理解其实现细节至关重要：

```
1  private boolean checkFinishId(Tuple tup, TupleType type) {
2      Object id = tup.getValue(0);
3      boolean failed = false;
4
5      synchronized(_tracked) {
6          TrackingInfo track = _tracked.get(id);
7          try {
8              if(track!=null) {
9                  boolean delayed = false;
10                 if(_idStreamSpec==null && type == TupleType.COORD || _idStreamSpec!=null &&
                       type==TupleType.ID) {
11                     track.ackTuples.add(tup);
12                     delayed = true;
13                 }
14                 if(track.failed) {
15                     failed = true;
16                     for(Tuple t: track.ackTuples) {
17                         _collector.fail(t);
18                     }
19                     _tracked.remove(id);
20                 } else if(track.receivedId
21                         && (_sourceArgs.isEmpty() ||
22                              track.reportCount==_numSourceReports &&
23                              track.expectedTupleCount == track.receivedTuples)){
24                     if(_delegate instanceof FinishedCallback) {
25                         ((FinishedCallback)_delegate).finishedId(id);
26                     }
27                     if(!(_sourceArgs.isEmpty() || type!=TupleType.REGULAR)) {
28                         throw new IllegalStateException("Coordination condition met
                               on a non-coordinatingtuple. Should be impossible");
29                     }
30                     Iterator<Integer> outTasks = _countOutTasks.iterator();
31                     while(outTasks.hasNext()) {
32                         int task = outTasks.next();
33                         int numTuples = get(track.taskEmittedTuples, task, 0);
```

```
34                      _collector.emitDirect(task, Constants.COORDINATED_STREAM_ID, tup, new
                            Values(id,numTuples));
35                  }
36                  for(Tuple t: track.ackTuples) {
37                      _collector.ack(t);
38                  }
39                  track.finished = true;
40                  _tracked.remove(id);
41              }
42              if(!delayed && type!=TupleType.REGULAR) {
43                  if(track.failed) {
44                      _collector.fail(tup);
45                  } else {
46                      _collector.ack(tup);
47                  }
48              }
49          } else {
50              if(type!=TupleType.REGULAR) _collector.fail(tup);
51          }
52      } catch(FailedException e) {
53          LOG.error("Failed to finish batch", e);
54          for(Tuple t: track.ackTuples) {
55              _collector.fail(t);
56          }
57          _tracked.remove(id);
58          failed = true;
59      }
60  }
61  return failed;
62 }
```

- 在第10~13行中，如果收到了控制消息，则将其放入变量ackTuples中，并在事务处理结束后统一进行Ack操作，或者在事务处理失败后统一进行Fail操作。
- 在第14~19行中，当认为事务已经失败时，将对收到的控制消息进行Fail操作，并将该事务尝试从跟踪列表中去掉。
- 在第20~41行中，判断事务是否已经处理结束并进行相应的处理。第20~23行用来判断是否已经完成了对事务的处理，具体条件如下。当满足这些条件时，表示该节点已经收全一个事务的所有消息，可以对事务进行后处理了。这是一个非常关键的时间点。
- 第20行的条件的含义为：若为事务提交Bolt节点，并且收到了从协调Spout发送来的事务提交消息。若为非事务提交Bolt节点，该条件默认为true。
- 第21~23行条件的含义为：该节点没有协调消息输入，或者该节点的消息源均向该节点发送了协调消息。系统中的协调Spout节点并不适合用CoordinatedBolt进行封装，所以事务Topology中的消息发送节点为最开始的CoordinatedBolt节点，再没有其他节点会向其发送协调消息了。但是消息发送节点只从协调Spout接收消息，一条消息即表示一个事务，因此其处理较为简单，当其收到一条消息后即认为已经收到了属于该事务的全部消息，并且满足向下游节点发送协调消息的条件。

- 在第24~26行中，如果用户的Bolt实现了FinishedCallback接口，此时将调用其finishId方法，即用户的finishBatch方法。此时，保证了Storm已经接收到了属于该事务的所有消息，于是可以调用finishBatch来对事务进行后处理。用户在实现IBatchBolt时，需要实现finishBatch方法。CoordinatedBolt保证一个Bolt节点收全了一个事务的所有消息后，finishBatch方法才会被调用。实际上，CoordinatedBolt可以用在非事务的批处理环境下。
- 在第27~29行中，若收到的消息为普通的消息，但是该Bolt此时已经认为收到了所有的数据，那么就进入了异常状态。
- 第30~35行用于向下游的节点发送协调消息，这样下游节点便可以用来判断事务是否可以结束。此时，整个Topology就通过协调消息串联了起来。注意第34行在调用emitDirect方法时以tup为标记，这也表示对该控制消息进行跟踪。
- 第36~38行用于对所有的控制消息进行Ack操作。
- 第42~48行作者认为是不需要的。delayed变量的默认值为false，并且只有当收到控制消息时才会被设置为true。而在这段代码中，要求收到的消息为控制信息，同时变量delayed为false，这是不能到达的。此外，如果前面已经对控制信息进行了Ack或Fail操作，那么再进行该种操作也会导致Storm的Ack系统出现问题。Storm在Trident的实现中对其进行了更正和优化。
- 第52~58行是异常处理，进行与第14~19行相类似的操作。

目前，我们分析了CoordinatedBolt中的大部分实现，其余代码与Topology的静态结构有关，即一个Bolt将从哪些Bolt接收协调数据，并向哪些Bolt发送协调数据，这将在15.7节中进一步讨论。

CoordinatedBolt是事务Topology的核心，http://xumingming.sinaapp.com/811 /twitter-storm-code-analysis-coordinated-bolt/包含了一些额外的分析。Storm项目的作者对此也有一些想法，具体请参考如下链接：https://github.com/nathanmarz/storm/issues/118。

15.5 分区的事务类型

分区的事务类型是对事务处理的进一步抽象，它可以认为是提供给用户的事务类型，Storm平台本身并不识别这种类型，而是通过将其适配为系统中的默认事务类型。故其接口与前面介绍的有很大不同，而Storm通过工具类将这些接口适配为ItransactionalSpout接口。这是Storm中常用的技巧，它简化了平台的设计，同时为平台提供了极大的扩展性。该技巧的模式为首先定义一个接口，然后通过一个执行类进行接口的适配。

本节首先介绍分区事务类型的接口，然后介绍其适配执行类。

15.5.1 分区的事务Spout接口

分区的事务的Spout是对基础事务的Spout的进一步抽象，用户可以通过它来完成对事务数据的分区处理。

首先，我们先来看一下IPartitionedTransactionalSpout接口的定义及分析：

```
public interface IPartitionedTransactionalSpout<T> extends IComponent {
    public interface Coordinator {
        /**
         * Return the number of partitions currently in the source of data. The idea is
         * is that if a new partition is added and a prior transaction is replayed, it doesn't
         * emit tuples for the new partition because it knows how many partitions were in
         * that transaction.
         */
        int numPartitions();

        /**
         * Returns true if its ok to emit start a new transaction, false otherwise (will skip
             thistransaction).
         *
         * You should sleep here if you want a delay between asking for the next transaction (this willbe
             called
         * repeatedly in a loop).
         */
        boolean isReady();

        void close();
    }

    public interface Emitter<X> {
        /**
         * Emit a batch of tuples for a partition/transaction that's never been emitted before.
         * Return the metadata that can be used to reconstruct this partition/batch in the future.
         */
        X emitPartitionBatchNew(TransactionAttempt tx, BatchOutputCollector collector, int partition,
            X lastPartitionMeta);

        /**
         * Emit a batch of tuples for a partition/transaction that has been emitted before, using
         * the metadata created when it was first emitted.
         */
        void emitPartitionBatch(TransactionAttempt tx, BatchOutputCollector collector, int partition,
             X partitionMeta);
        void close();
    }

    Coordinator getCoordinator(Map conf, TopologyContext context);
    Emitter<T> getEmitter(Map conf, TopologyContext context);
}
```

这个接口与ITransactionalSpout接口有很大的不同，代表了另外一种抽象方式，它在协调Spout中储存的关于事务的元信息是每个事务所含分区的数目。每一个事务中与各个分区相关的元数据由emitPartitionBatchNew方法指定，该方法是在消息发送节点上被调用的。下面简要介绍该接口中涉及的几个方法。

- numPartitions方法会在ITransactionalSpout.initializeTransaction方法中调用，返回每一个事务所包含的分区数目。

- isReady方法与ITransactionalSpout.isReady方法的含义相同，表示数据是否已经准备好以便开始一个新的事务。
- emitPartitionBatchNew方法以及emitPartitionBatch方法会在ITransactionalSpout.emitBatch中被调用。emitPartitionBatchNew方法会在一个事务首次进行尝试时被调用，它会基于当前分区以及事务，计算获得该分区所对应的元数据。
- emitPartitionBatch会在重传时被调用。当然，在具体的实现中，emitPartitionBatchNew方法可以基于emitPartitionBatch方法来实现。区别在于，emitPartitionBatch方法知道该分区在当前事务上面的元数据是什么，而emitPartitionBatchNew则需要通过计算来得到该元数据。

15.5.2 分区的事务Spout的执行器

PartitionedTransactionalSpoutExecutor类实现了ITransactionalSpout接口，请注意它的元数据的类型为Integer类型，表示分区数目。

这个类实现可以被认为是学习接口ITransactionalSpout及事务Topology的一个实例。

PartitionedTransactionalSpoutExecutor中含有一个IPartitionedTransactionalSpout的代理_spout成员变量，ITransactionalSpout中定义的方法将通过调用_spout中相应的方法来实现。该类的实现代码如下：

```
public class PartitionedTransactionalSpoutExecutor implements ITransactionalSpout<Integer> {
    IPartitionedTransactionalSpout _spout;

    public PartitionedTransactionalSpoutExecutor(IPartitionedTransactionalSpout spout) {
        _spout = spout;
    }

    public IPartitionedTransactionalSpout getPartitionedSpout() {
        return _spout;
    }


    @Override
    public ITransactionalSpout.Coordinator getCoordinator(Map conf, TopologyContext context) {
        return new Coordinator(conf, context);
    }

    @Override
    public ITransactionalSpout.Emitter getEmitter(Map conf, TopologyContext context) {
        return new Emitter(conf, context);
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        _spout.declareOutputFields(declarer);
    }
```

```
    @Override
    public Map<String, Object> getComponentConfiguration() {
        return _spout.getComponentConfiguration();
    }

}
```

接下来讨论协调Spout的相关接口的实现如下：

```
1  class Coordinator implements ITransactionalSpout.Coordinator<Integer> {
2      private IPartitionedTransactionalSpout.Coordinator _coordinator;
3
4      public Coordinator(Map conf, TopologyContext context) {
5          _coordinator = _spout.getCoordinator(conf, context);
6      }
7
8      @Override
9      public Integer initializeTransaction(BigInteger txid, Integer prevMetadata) {
10         return _coordinator.numPartitions();
11     }
12
13     @Override
14     public boolean isReady() {
15         return _coordinator.isReady();
16     }
17
18     @Override
19     public void close() {
20         _coordinator.close();
21     }
22 }
```

- 第4~6行在构造函数中得到IPartitionedTransactionalSpout.Coordinator。
- 第9~11行实现initializeTransaction方法，它将返回分区的数目。
- 第14~16行实现isReady方法，调用_coordinator对象的isReady方法。在协调Spout中，当产生新的事务尝试时，会将其对应分区的数目作为元数据。

下面来看消息发送节点及相关接口的实现，相关代码如下：

```
1  class Emitter implements ITransactionalSpout.Emitter<Integer> {
2      private IPartitionedTransactionalSpout.Emitter _emitter;
3      private TransactionalState _state;
4      private Map<Integer, RotatingTransactionalState> _partitionStates = new HashMap<Integer,
            RotatingTransactionalState>();
5      private int _index;
6      private int _numTasks;
7
8      public Emitter(Map conf, TopologyContext context) {
9          _emitter = _spout.getEmitter(conf, context);
10         _state = TransactionalState.newUserState(conf, (String)
           conf.get(Config.TOPOLOGY_TRANSACTIONAL_ID), getComponentConfiguration());
11     _index =context. getThisTaskIndex();
```

```
12         _numTasks = context.getComponentTasks(context.getThisComponentId()).size();
13     }
14 
15     @Override
16     public void emitBatch(final TransactionAttempt tx, final Integer partitions,
17                           final BatchOutputCollector collector) {
18         for(int i=_index; i < partitions; i+=_numTasks) {
19             if(!_partitionStates.containsKey(i)) {
20                 _partitionStates.put(i, new RotatingTransactionalState(_state, "" + i));
21             }
22             RotatingTransactionalState state = _partitionStates.get(i);
23             final int partition = i;
24             Object meta = state.getStateOrCreate(tx.getTransactionId(),
25                      new RotatingTransactionalState.StateInitializer() {
26                 @Override
27                 public Object init(BigInteger txid, Object lastState) {
28                     return _emitter.emitPartitionBatchNew(tx, collector, partition, lastState);
29                 }
30             });
31             // it's null if one of:
32             //a) a later transaction batch was emitted before this, so we should skip this batch
33             //b) if didn't exist and was created (in which case the StateInitializer
                   was invoked and
34             // it was emitted
35             if(meta!=null) {
36                 _emitter.emitPartitionBatch(tx, collector, partition, meta);
37             }
38         }
39     }
40 
41     @Override
42     public void cleanupBefore(BigInteger txid) {
43         for(RotatingTransactionalState state: _partitionStates.values()) {
44             state.cleanupBefore(txid);
45         }
46     }
47 
48     @Override
49     public void close() {
50         _state.close();
51         _emitter.close();
52     }
53 }
```

- 第3行定义的_state为TransactionalState类型，它所对应的数据被放在user子目录下。根据前面的讨论，协调Spout节点所对应的元数据存放在coordinator子目录下。此处的目录是指ZooKeeper中的路径。
- 第4行中的_partitionStates为一个哈希表，每个分区都对应了一个RotatingTransactionalState对象。注意在初始化RotatingTransactionalState时，并不要求进行事务元数据的强序检查。假设我们有两个分区P_1和P_2，目前有两个活跃的事务序号为$txid_1$和$txid_2$，且Topology中SpoutId为SPOUT，那么在运行过程中将会产生如下的目录结构：

```
/transactional/SPOUT/coordinator/txid₁
/transactional/SPOUT/coordinator/txid₂
/transactional/SPOUT/user/P₁/txid₁
/transactional/SPOUT/user/P₁/txid₂
/transactional/SPOUT/user/P₂/txid₁
/transactional/SPOUT/user/P₂/txid₂
```

即每个分区都会对应同样的事务序列，分区P_1和P_2都同样含有$txid_1$和$txid_2$事务序列。仔细揣摩分区与事务的对应关系是理解这个类的关键。

- 第5行定义的_index表示当前消息发送节点在所有消息发送节点中的索引，它是从0开始的。例如，消息发送节点的并行度为10，那么系统中将存在10个消息发送节点，每个节点都会被唯一地赋予一个0~9之间的标号来进行区分。TaskId是Topology给所有Task赋予的编号，即便Task重启这个编号也不会改变。而 _index索引实际上是基于TaskId的，因此_index号码也是稳定的，它可通过调用getComponentTasks方法获得。
- 第6行定义的_numTasks表示消息发送节点的并行度。
- 第16~39行是emitBatch方法的实现，它将会调用emitPartitionBatchNew以及emitPartitionBatch方法。
- 第18行在消息发送节点上对分区进行分配。例如，有10个分区，以及2个节点E_0、E_1，那么E_0将处理P_0、P_2、P_4、P_6、P_8，而E_2将处理P_1、P_3、P_5、P_7、P_9，其中Pi代表第i个分区。于是，当分区数目小于消息发送节点的数目时，将有一部分节点处于空闲状态。
- 在第19~21行中，若分区所对应的RotatingTransactionalState对象不存在，则创建该对象。在ZooKeeper中，子目录为分区的编号。
- 第24~30行用于为当前的分区创建元数据。getStateOrCreate方法的第二个参数需要传入一个实现了接口StateInitializer的对象。第25~29行定义了一个内部类并将其作为参数传入。这里的init方法将调用emitPartitionBatchNew方法，可以保证它只有在处理新事务时才会被调用。

  getStateOrCreate方法在以下两种情况下可能返回空。

  - 新的事务已被初始化并通过emitPartitionBatchNew方法发送出去。
  - 比当前事务更大的事务已经被发送出去了，并且该事务是基于当前事务前面的事务产生的，于是我们需要忽略当前的事务，详情请参考4.5节。

  当返回的元数据不为空时，将调用emitPartitionBatch方法，表示对当前事务进行重传。
- 第42~46行定义了cleanupBefore方法，它会根据已经提交的事务序号对元数据进行清理。

由于不同的消息发送节点所负责的事务分区是不同的，因此它们对ZooKeeper的数据操作是没有冲突的。通过这个类的实现可以得知，并不仅仅是协调Spout节点可以产生和维护元数据，如果设计合理，Bolt端同样也可以完成类似的事情。

15.6 分区的模糊事务 Spout

目前，Storm只支持具有分区功能的模糊事务类型的Spout，它们主要用于读取Kafka队列中的数据，该类型也是系统中最复杂的Spout类型。其实现技术与分区的基本事务的Spout相似：先定义一个用户接口，然后再定义一个类进行接口适配并执行。

15.6.1 分区的模糊事务Spout的接口

接口IOpaquePartitionedTransactionalSpout与ITransactionalSpout的区别也较大，它实际上并不会在协调Spout中存储任何元数据，其代码如下：

```
public interface IOpaquePartitionedTransactionalSpout<T> extends IComponent {
    public interface Coordinator {
        /**
         * Returns true if its ok to emit start a new transaction, false otherwise (will skip this
transaction).
         *
         * You should sleep here if you want a delay between asking for the next transaction (this will
be called
         * repeatedly in a loop).
         */
        boolean isReady();
        void close();
    }

    public interface Emitter<X> {
        /**
         * Emit a batch of tuples for a partition/transaction.
         *
         * Return the metadata describing this batch that will be used as lastPartitionMeta
         * for defining the parameters of the next batch.
         */
        X emitPartitionBatch(TransactionAttempt tx, BatchOutputCollector collector, int partition, X
lastPartitionMeta);
        int numPartitions();
        void close();
    }

    Emitter<T> getEmitter(Map conf, TopologyContext context);
    Coordinator getCoordinator(Map conf, TopologyContext context);
}
```

协调Spout接口中只定义了isReady方法，用于表示数据源否准备好以及是否开始一个新的事务。

相比于IPartitionedTransactionalSpout接口，IOpaquePartitionedTransactionalSpout接口中numParitions方法被放入了Emitter接口里面。

emitPartitionBatch 的 参 数 列 表 与 IPartitionedTransactionalSpout 中 的 emitPartitionBatchNew方法很类似。

15.6.2　模糊的事务Spout执行器

OpaquePartitionedTransactionalSpoutExecutor类实现了ICommitterTransactionalSpout接口，该接口继承自ITransactionalSpout接口，并添加了commit接口方法。该类的消息发送接口较为复杂，本文将对其进行单独分析。

OpaquePartitionedTransactionalSpoutExecutor类的实现代码如下：

```
1 public class OpaquePartitionedTransactionalSpoutExecutor implements ICommitterTransactionalSpout
      <Object> {
2     IOpaquePartitionedTransactionalSpout _spout;
3
4     public class Coordinator implements ITransactionalSpout.Coordinator<Object> {
5         IOpaquePartitionedTransactionalSpout.Coordinator _coordinator;
6
7         public Coordinator(Map conf, TopologyContext context) {
8             _coordinator = _spout.getCoordinator(conf, context);
9         }
10
11        @Override
12        public Object initializeTransaction(BigInteger txid, Object prevMetadata) {
13            return null;
14        }
15
16        @Override
17        public boolean isReady() {
18            return _coordinator.isReady();
19        }
20
21        @Override
22        public void close() {
23            _coordinator.close();
24        }
25    }
26
27
28     public OpaquePartitionedTransactionalSpoutExecutor(IOpaquePartitionedTransactional Spout
           spout) {
29         _spout = spout;
30     }
31
32     @Override
33    public ITransactionalSpout.Coordinator<Object> getCoordinator(Map conf, Topology Context
          context) {
34        return new Coordinator(conf, context);
35    }
36
37    @Override
38     public ICommitterTransactionalSpout.Emitter getEmitter(Map conf, TopologyContext context) {
39         return new Emitter(conf, context);
40    }
41
```

```
42     @Override
43     public void declareOutputFields(OutputFieldsDeclarer declarer) {
44         _spout.declareOutputFields(declarer);
45     }
46
47     @Override
48     public Map<String, Object> getComponentConfiguration() {
49         return _spout.getComponentConfiguration();
50     }
51 }
```

- 第1行说明该类实现了ICommitterTransactionalSpout接口，表明在消息发送节点中需要实现commit回调方法。
- 第12~14行定义的initializeTransaction方法的返回值为空，表示在协调Spout中不存储任何元数据。其他方法都是比较直观的，读者可自行分析。

下面主要分析其消息发送接口的实现，相关代码如下：

```
1  public class Emitter implements ICommitterTransactionalSpout.Emitter {
2      IOpaquePartitionedTransactionalSpout.Emitter _emitter;
3      TransactionalState _state;
4      TreeMap<BigInteger, Map<Integer, Object>> _cachedMetas = new TreeMap<BigInteger,
           Map<Integer,Object>>();
5      Map<Integer, RotatingTransactionalState> _partitionStates = new HashMap<Integer,
           RotatingTransactionalState>();
6      int _index;
7      int _numTasks;
8
9      public Emitter(Map conf, TopologyContext context) {
10         _emitter = _spout.getEmitter(conf, context);
11         _index = context.getThisTaskIndex();
12         _numTasks = context.getComponentTasks(context.getThisComponentId()).size();
13         _state = TransactionalState.newUserState(conf, (String) conf.get(Config.TOPOLOGY_
               TRANSACTIONAL_ID), getComponentConfiguration());
14         List<String> existingPartitions = _state.list("");
15         for(String p: existingPartitions) {
16             int partition = Integer.parseInt(p);
17             if((partition - _index) % _numTasks == 0) {
18                 _partitionStates.put(partition, new RotatingTransactionalState(_state, p));
19             }
20         }
21     }
22
23     @Override
24     public void emitBatch(TransactionAttempt tx, Object coordinatorMeta, BatchOutputCollector
          collector) {
25         Map<Integer, Object> metas = new HashMap<Integer, Object>();
26         _cachedMetas.put(tx.getTransactionId(), metas);
27         int partitions = _emitter.numPartitions();
28         Entry<BigInteger, Map<Integer, Object>> entry = _cachedMetas.lowerEntry
               (tx.getTransactionId());
29         Map<Integer, Object> prevCached;
```

```
30          if(entry!=null) {
31              prevCached = entry.getValue();
32          } else {
33               prevCached = new HashMap<Integer, Object>();
34          }
35
36          for(int i=_index; i < partitions; i+=_numTasks) {
37              RotatingTransactionalState state = _partitionStates.get(i);
38              if(state==null) {
39                  state = new RotatingTransactionalState(_state, "" + i);
40                  _partitionStates.put(i, state);
41              }
42              state.removeState(tx.getTransactionId());
43              Object lastMeta = prevCached.get(i);
44              if(lastMeta==null) lastMeta = state.getLastState();
45              Object meta = _emitter.emitPartitionBatch(tx, collector, i, lastMeta);
46              metas.put(i, meta);
47          }
48      }
49
50      @Override
51      public void cleanupBefore(BigInteger txid) {
52          for(RotatingTransactionalState state: _partitionStates.values()) {
53              state.cleanupBefore(txid);
54          }
55      }
56
57      @Override
58      public void commit(TransactionAttempt attempt) {
59          BigInteger txid = attempt.getTransactionId();
60          Map<Integer, Object> metas = _cachedMetas.remove(txid);
61          for(Integer partition: metas.keySet()) {
62              Object meta = metas.get(partition);
63              _partitionStates.get(partition).overrideState(txid, meta);
64          }
65      }
66
67      @Override
68      public void close() {
69          _emitter.close();
70      }
71  }
```

在这个类中，我们需要重点理解的是分区的元数据的产生及更新。

- 第4行定义了类成员变量_cachedMetas，它的键为事务序号，值同样也是一个哈希表。该哈希表的键为分区号，值为此分区的元数据，即总的映射关系为事务序号->分区号 ->分区元数据。
- 第9~21行定义了消息发送节点的构造方法。第15~20行将已经存放在ZooKeeper中的数据加载到_partitionStates成员变量中。第17行与第36行计算得到的分区与节点的对应关系是相同的，当Topology启动后，哪些Emitter节点处理哪些分区是固定的。

- 第24~48行实现了emitBatch方法。这里并不会去区分事务究竟是新事务还是重传事务。
- 第25行定义了一个全新的哈希表类型的元数据对象，用来存储当前事务下每个分区所对应的元数据。
- 第27行调用numPartitions方法获得当前的分区数目。
- 第28~34行试图获取前一个事务所对应的元数据。
- 第36~47行对当前消息发送节点上的每一个分区进行处理。第42行将ZooKeeper中的数据清理掉，表示当前事务正在被处理。
- 第45行调用emitPartitionBatch方法，并将该分区所对应的数据放于元数据中。值得注意的是，在emitBatch方法中，分区所对应的元数据并没有被写回到ZooKeeper中，它们在调用commit方法时被写回。
- 第58~65行定义了commit方法，该方法只有在收到了从协调Spout节点发送到消息提交流上的消息时才会被调用。收到该消息意味着事务在系统中已经被成功处理了，此时我们将事务序号所对应的元数据从_cachedMetas中删除，并将其值更新至ZooKeeper中。

通过分析可以看到，每个分区所对应的元数据只有在事务成功处理之后才会被存放在ZooKeeper中，这与其他类型Spout的实现是不同的。

ITransactionalSpout及IPartitionedTransactionalSpout都要求每个事务所对应的数据要前后完全一致，而IOpaqueTransactionalSpout并不要求这点，其中每个事务所对应的数据都是可以变化的，但要求事务之间的数据没有重复。

15.7 事务 Topology 的构建器

TransactionalTopologyBuilder用于方便用户构建事务类型的Topology，它将完成节点的系统流的添加及接收设置。例如，它将为协调Spout节点定义两个流：事务流和事务提交流。

TransactionalTopologyBuilder类提供类似于TopologyBuilder类的方法，用来设置Spout和Bolt节点。它的复杂性在于如何将ITransactionalSpout中定义的协调Spout节点及消息发送节点部署到Topology中，映射到基础的Spout和Bolt节点上，并描述它们的依赖关系。

为了便于讨论，本节将该类分成几部分进行讨论。下面首先分析其构造函数以及成员变量。

15.7.1 构建器的构造函数及成员变量

TransactionalTopologyBuilder类的定义如下，从其构造函数可以看出，事务Topology中只能存在一种类型的Spout节点。在构造函数中，需要传入一个ITransactionalSpout类型的Spout对象。详细分析如下：

```
public class TransactionalTopologyBuilder {
    String _id;
    String _spoutId;
    ITransactionalSpout _spout;
    Map<String, Component> _bolts = new HashMap<String, Component>();
```

```
    Integer _spoutParallelism;
    List<Map> _spoutConfs = new ArrayList();

    // id is used to store the state of this transactionalspout in zookeeper
    // it would be very dangerous to have 2 topologies active with the same id in the same cluster
    public TransactionalTopologyBuilder(String id, String spoutId, ITransactionalSpout spout, Number
        spoutParallelism) {
        _id = id;
        _spoutId = spoutId;
        _spout = spout;
        _spoutParallelism = (spoutParallelism == null) ? null : spoutParallelism.intValue();
    }

    public TransactionalTopologyBuilder(String id, String spoutId, ITransactionalSpout spout) {
        this(id, spoutId, spout, null);
    }

    public TransactionalTopologyBuilder(String id, String spoutId, IPartitionedTransactionalSpout
       spout, Number spoutParallelism) {
        this(id, spoutId, new PartitionedTransactionalSpoutExecutor(spout), spoutParallelism);
    }

    public TransactionalTopologyBuilder(String id, String spoutId, IPartitionedTransactionalSpout
       spout) {
        this(id, spoutId, spout, null);
    }

    public TransactionalTopologyBuilder(String id, String spoutId,IOpaquePartitionedTransactional
        Spout  spout, Number spoutParallelism) {
        this(id, spoutId, new OpaquePartitionedTransactionalSpoutExecutor(spout), spoutParallelism);
    }

    public TransactionalTopologyBuilder(String id, String spoutId,IOpaquePartitionedTransactional
        Spout spout) {
        this(id, spoutId, spout, null);
    }

    public SpoutDeclarer getSpoutDeclarer() {
        return new SpoutDeclarerImpl();
    }
}
```

TransactionalTopologyBuilder的构造函数接收如下3类对象。

- ITransactionalSpout：基础事务类型的 Spout实现。它设置了基本的类成员变量，例如TopologyId、SpoutId和并行度等。
- IPartitionedTransactionalSpout：使用PartitionedTransactionalSpoutExecutor进行封装执行，调用参数为ITransactionalSpout类型的构造函数。
- IOpaquePartitionedTransactionalSpout：使用OpaquePartitionedTransactionalSpoutExecutor进行包装执行，然后调用参数为ITransactionalSpout类型的构造函数。

可以看出，系统中唯一支持的类型为ITransactionalSpout类型，其他两种类型为适配的结果。

下面简要分析一下该类的成员变量。

- _id：用来设置TOPOLOGY_TRANSACTIONAL_ID，并将其作为ZooKeeper中元数据目录结构的一部分。
- _spoutId：Spout的名字。由_spoutId 与 /coordinator连接构成。
- _spout：具体的Spout对象，类型为ITransactionalSpout。
- _bolts：Topology中的Bolt对象。
- _spoutParallelism：实际上为消息发送节点的数目，其中协调Spout节点只能有一个。
- _spoutConfs：Spout对象的配置项。

15.7.2　设置Bolt对象

该类中提供了较多的重载方法来设置系统中的Bolt节点，相关代码如下：

```
public BoltDeclarer setBolt(String id, IBatchBolt bolt) {
    return setBolt(id, bolt, null);
}

public BoltDeclarer setBolt(String id, IBatchBolt bolt, Number parallelism) {
    return setBolt(id, new BatchBoltExecutor(bolt), parallelism, bolt instanceof ICommitter);
}

public BoltDeclarer setCommitterBolt(String id, IBatchBolt bolt) {
    return setCommitterBolt(id, bolt, null);
}

public BoltDeclarer setCommitterBolt(String id, IBatchBolt bolt, Number parallelism) {
    return setBolt(id, new BatchBoltExecutor(bolt), parallelism, true);
}

public BoltDeclarer setBolt(String id, IBasicBolt bolt) {
    return setBolt(id, bolt, null);
}

public BoltDeclarer setBolt(String id, IBasicBolt bolt, Number parallelism) {
    return setBolt(id, new BasicBoltExecutor(bolt), parallelism, false);
}

private BoltDeclarer setBolt(String id, IRichBolt bolt, Number parallelism, boolean committer) {
    Integer p = null;
    if(parallelism!=null) p = parallelism.intValue();
    Component component = new Component(bolt, p, committer);
    _bolts.put(id, component);
    return new BoltDeclarerImpl(component);
}
```

setBolt方法有很多重载，它们主要是为了区分以下几种情况。

- IRichBolt：事务Topology的基本类型。
- IBasicBolt：使用BasicBoltExecutor进行封装执行，并调用第一种类型的构造函数。
- IBatchBolt：使用BatchBoltExecutor进行封装执行，并调用第一种类型的构造函数。

事务提交Bolt其实为IBatchBolt类型，同时实现了ICommitter接口的Bolt对象，Storm在构建Topology的流接收关系时会有所不同。例如，在事务Topology中，实现ICommitter接口的Bolt将接收事务提交消息。另外可以看出，在事务Topology中执行的Bolt均为IRichBolt类型，其他类型的Bolt节点也会被适配成IRichBolt。另外，BatchBoltExecutor以及BasicBoltExecutor会对输入的消息进行Ack操作，若用户想要实现IRichBolt接口，则需要自己完成对消息的Ack操作。对消息进行Ack操作是Ack框架可以正常工作的基础。

15.7.3　构建Topology

本节将讨论如何来构建事务Topology，相关代码如下：

```
1 public TopologyBuilder buildTopologyBuilder() {
2     String coordinator = _spoutId + "/coordinator";
3     TopologyBuilder builder = new TopologyBuilder();
4     SpoutDeclarer declarer = builder.setSpout(coordinator, new TransactionalSpoutCoordinator
          (_spout));
5     for(Map conf: _spoutConfs) {
6         declarer.addConfigurations(conf);
7     }
8     declarer.addConfiguration(Config.TOPOLOGY_TRANSACTIONAL_ID, _id);
9
10    BoltDeclarer emitterDeclarer =
11            builder.setBolt(_spoutId,
12                              new CoordinatedBolt(new TransactionalSpoutBatchExecutor(_spout),
13                                                  null,
14                                                  null),
15                                _spoutParallelism)
16            .allGrouping(coordinator, TransactionalSpoutCoordinator.TRANSACTION_BATCH_STREAM_ID)
17             .addConfiguration(Config.TOPOLOGY_TRANSACTIONAL_ID, _id);
18     if(_spout instanceof ICommitterTransactionalSpout) {
19         emitterDeclarer.allGrouping(coordinator, TransactionalSpoutCoordinator.TRANSACTION_
               COMMIT_STREAM_ID);
20     }
21     for(String id: _bolts.keySet()) {
22         Component component = _bolts.get(id);
23         Map<String, SourceArgs> coordinatedArgs = new HashMap<String, SourceArgs>();
24         for(String c: componentBoltSubscriptions(component)) {
25             coordinatedArgs.put(c, SourceArgs.all());
26         }
27
28         IdStreamSpec idSpec = null;
29         if(component.committer) {
30             idSpec = IdStreamSpec.makeDetectSpec(coordinator, TransactionalSpoutCoordinator.
                   TRANSACTION_COMMIT_STREAM_ID);
31         }
32         BoltDeclarer input = builder.setBolt(id,
33                                              new CoordinatedBolt(component.bolt,
34                                                                  coordinatedArgs,
35                                                                  idSpec),
36                                              component.parallelism);
```

```
37         for(Map conf: component.componentConfs) {
38             input.addConfigurations(conf);
39         }
40         for(String c: componentBoltSubscriptions(component)) {
41             input.directGrouping(c, Constants.COORDINATED_STREAM_ID);
42         }
43         for(InputDeclaration d: component.declarations) {
44             d.declare(input);
45         }
46         if(component.committer) {
47             input.allGrouping(coordinator, TransactionalSpoutCoordinator.TRANSACTION_COMMIT_
                   STREAM_ID);
48         }
49     }
50     return builder;
51 }
```

理解这部分代码对于理解事务Topology是非常重要的，它定义了各个节点之间的协作方式。

- 第3行创建了一个基本的TopologyBuilder对象。
- 第4行设置Topology中的协调Spout节点。它利用TransactionalSpoutCoordinator对ITransationalSpout中的协调Spout接口进行封装执行。根据之前的分析可以知道，TransactionalSpoutCoordinator实现了IRichSpout接口。
- 第10~15行用于设置Topology中的消息发送节点。首先，利用TransactionalSpoutBatchExecutor对ITransactionalSpout中的消息发送接口进行封装，然后，再利用CoordinatedBolt进行进一步的封装。在事务Topology中执行的所有Bolt均为CoordinatedBolt类型。
- 第16~17行设置消息发送Bolt通过全局分组的方式接收协调Spout中的事务流。
- 在第18~20行中，如果Committer实现了ICommitterTransactionalSpout接口，那么消息发送节点将通过全局分组的方式接收协调Spout中的事务提交流。目前，这部分是为IOpaquePartitionedTransactionalSpout设计的。
- 第21~49行设置Topology中的Bolt节点，这些节点由用户定义。
  - 第22~26行获得每个Bolt的上游节点组件名称，为确定CoordinatedBolt接收哪些协调消息流做准备。
  - 在第28~31行中，若该Bolt实现了ICommiter接口，将设置IdStreamSpec，含义为协调Spout的消息提交流。
  - 第32~36行设置Bolt对象，它们均是利用CoordinatedBolt进行封装执行的。setBolt函数将返回了一个BoltDeclarer对象，用于完成由用户定义的流的接收关系。input变量目前为类TopologyBuilder中定义的BoltGetter对象。
  - 第40~42行用于将Bolt的每一个上游节点均设置为直接分组到协调消息流上。
  - 第43~45行是用户自定义的分组方式，源自于用户逻辑。
  - 在第46~48行中，如果Bolt实现了ICommiter接口，则将其设置为全局分组到协调Spout的事务提交流。

15.7.4　输入流声明器

在本节讨论中，我们将各个组件是如何声明其输入流的。

内部类Component主要被事务Topology使用，其代码如下：

```
private static class Component {
    public IRichBolt bolt;
    public Integer parallelism;
    public List<InputDeclaration> declarations = new ArrayList<InputDeclaration>();
    public List<Map> componentConfs = new ArrayList<Map>();
    public boolean committer;

    public Component(IRichBolt bolt, Integer parallelism, boolean committer) {
        this.bolt = bolt;
        this.parallelism = parallelism;
        this.committer = committer;
    }
}
```

成员变量declarations是List<InputDeclaration>类型的，表示该组件将接收哪些流，其中接口InputDeclaration的定义如下：

```
private static interface InputDeclaration {
    void declare(InputDeclarer declarer);
    String getComponent();
}
```

在上述代码中，declare方法需要传入InputDeclarer对象，InputDeclarer接口在前面章节中介绍过，目前使用类TopologyBuilder的内部类BoltGetter作为其实现。

最后，我们看一下BoltDeclarerImpl的实现。为节约篇幅，仅以域分组的实现为例进行介绍，其他分组方式的实现类似。该类的代码如下：

```
1 private class BoltDeclarerImpl extends BaseConfigurationDeclarer<BoltDeclarer> implements
       BoltDeclarer {
2     Component _component;
3
4      public BoltDeclarerImpl(Component component) {
5          _component = component;
6      }
7
8      @Override
9      public BoltDeclarer fieldsGrouping(final String component, final String streamId,
           final Fields fields) {
10       addDeclaration(new InputDeclaration() {
11           @Override
12           public void declare(InputDeclarer declarer) {
13               declarer.fieldsGrouping(component, streamId, fields);
14           }
15
16           @Override
```

```
17            public String getComponent() {
18                return component;
19            }
20        });
21        return this;
22    }
23    private void addDeclaration(InputDeclaration declaration) {
24        _component.declarations.add(declaration);
25    }
26
27    @Override
28    public BoltDeclarer addConfigurations(Map conf) {
29        _component.componentConfs.add(conf);
30        return this;
31    }
32 }
```

- 第10~19行实现了接口InputDeclaration。第12~14行实现了declare方法，其中调用了declarer的fieldsGrouping方法。目前，declarer实际上为BoltGetter类型。
- 第23~25行定义了addDeclaration方法，它将输入的declaration存入到Component对象中的链表declarations中。

在这个类的实现中，我们使用了内部类的概念以及较多的接口。内部类技术使得内部类函数可以直接访问外部类的变量，例如，传入的流和分组变量就必须被定义为final类型。

第 16 章 事务Topology示例

本章将介绍Storm Starter中的一个分区事务Topology的例子，读者可以结合前一章中对事务Topology的分析来学习这个例子。Storm Starter是与Storm一起发布的示例代码集合。

16.1 例子代码

本节将介绍该例子中各个组件的实现。它含有一个分区事务的Spout用于发送数据，同时含有局部计数节点用于局部计数，还有全局计数节点用于全局的计数、去重和存储。该实例只能在LocalCluster中运行，不能被部署到集群环境中。

16.1.1 分区的事务Spout

MemoryTransactionalSpout类实现了IPartitionedTransactionalSpout，它利用三个队列模拟的数据作为数据分区。由于该类主要用于Storm的内部测试，故这里已将与测试相关的代码去除。该类的实现代码如下：

```
1 public class MemoryTransactionalSpout implementsIPartitionedTransactionalSpout
     <MemoryTransactionalSpoutMeta> {
2       public static String TX_FIELD = MemoryTransactionalSpout.class.getName() + "/id";
3
4       private int _takeAmt;
5       private Fields _outFields;
6       private Map<Integer, List<List<Object>>> _initialPartitions;
7
8       public MemoryTransactionalSpout(Map<Integer, List<List<Object>>> partitions,
             Fields outFields, int takeAmt) {
9           _id = RegisteredGlobalState.registerState(partitions);
10          Map<Integer, Boolean> finished = Collections.synchronizedMap(new HashMap<Integer,
                Boolean>());
11          _takeAmt = takeAmt;
12          _outFields = outFields;
13          _initialPartitions = partitions;
14      }
15
16      @Override
17      public IPartitionedTransactionalSpout.Coordinator getCoordinator(Map conf, TopologyContext
```

```
        context) {
18          return new Coordinator();
19      }
20
21      @Override
22      public IPartitionedTransactionalSpout.Emitter<MemoryTransactionalSpoutMeta> getEmitter
            (Map conf, TopologyContext context) {
23          return new Emitter(conf);
24      }
25
26      @Override
27      public void declareOutputFields(OutputFieldsDeclarer declarer) {
28          List<String> toDeclare = new ArrayList<String>(_outFields.toList());
29          toDeclare.add(0, TX_FIELD);
30          declarer.declare(new Fields(toDeclare));
31      }
32
33      @Override
34      public Map<String, Object> getComponentConfiguration() {
35          Config conf = new Config();
36          conf.registerSerialization(MemoryTransactionalSpoutMeta.class);
37          return conf;
38      }
39
40      public void cleanup() {
41          RegisteredGlobalState.clearState(_id);
42      }
43
44      private Map<Integer, List<List<Object>>> getQueues() {
45          Map<Integer, List<List<Object>>> ret = (Map<Integer, List<List<Object>>>) Registered
                GlobalState.getState(_id);
46          if(ret!=null) return ret;
47          else return _initialPartitions;
48      }
49  }
```

- _initialPartitions为数据源，它的类型为一个哈希表，键为分区号，值为一个队列，代表分区的数据。RegisteredGlobalState类用于存储全局的数据，它将数据保存在一个静态的哈希表中，并在访问数据时进行同步。它的主要目标为模拟多个Emitter，并不具备实际的意义。第44~48行定义的getQueues方法用于获得数据源。该例子只能通过LocalCluster运行，而不能放在真正的集群中去运行。
- _takeAmt表示每个事务所对应的数据条数。
- MemoryTransactionalSpoutMeta为事务所对应的元数据，它在emitPartitionBatchNew方法中产生。该类需要由用户实现，相关的定义如下：

```
public class MemoryTransactionalSpoutMeta {
    int index;
    int amt;
}
```

- 第36行进行序列化注册。类MemoryTransactionalSpoutMeta的对象实例会被写入ZooKeeper中。
- 第27~31行定义的declareOutputFields方法需要将事务序号作为第1列声明，该步骤是必需的，也是初学者容易忘记的。

由于MemoryTransactionalSpout实现了IPartitionedTransactionalSpout接口，用户需要实现Coordinator和Emitter接口。

下面首先来看Coordinator接口的实现，其代码如下：

```
class Coordinator implements IPartitionedTransactionalSpout.Coordinator {

    @Override
    public int numPartitions() {
        return getQueues().size();
    }

    @Override
    public boolean isReady() {
        return true;
    }

    @Override
    public void close() {
    }
}
```

在上述代码中，numPartitions方法会返回分区的数目，isReady方法则返回true。

接下来看Emitter的接口实现：

```
1 lass Emitter implements IPartitionedTransactionalSpout.Emitter<MemoryTransactionalSpoutMeta> {
2     public Emitter(Map conf) {
3     }
4
5     @Override
6     public MemoryTransactionalSpoutMeta emitPartitionBatchNew(TransactionAttempt tx,
          BatchOutputCollector collector, int partition, MemoryTransactionalSpoutMeta
          lastPartitionMeta) {
7         int index;
8         if(lastPartitionMeta==null) {
9               index = 0;
10        } else {
11             index = lastPartitionMeta.index + lastPartitionMeta.amt;
12        }
13        List<List<Object>> queue = getQueues().get(partition);
14        int total = queue.size();
15        int left = total - index;
16        int toTake = Math.min(left, _takeAmt);
17
18      MemoryTransactionalSpoutMeta ret = new MemoryTransactionalSpoutMeta(index, toTake);
19      emitPartitionBatch(tx, collector, partition, ret);
```

```
20         return ret;
21     }
22 
23     @Override
24     public void emitPartitionBatch(TransactionAttempt tx, BatchOutputCollector collector, int
           partition, MemoryTransactionalSpoutMeta partitionMeta) {
25         List<List<Object>> queue = getQueues().get(partition);
26         for(int i=partitionMeta.index; i < partitionMeta.index + partitionMeta.amt; i++) {
27             List<Object> toEmit = new ArrayList<Object>(queue.get(i));
28             toEmit.add(0, tx);
29             collector.emit(toEmit);
30         }
31     }
32 
33     @Override
34     public void close() {
35     }
36 }
```


在上述代码中，第6~21行定义了emitPartitionBatchNew方法，该方法需要构建事务的元数据，在处理第一个事务时，lastPartitionMeta为空。index表示数据队列中的偏移量，toTake表示要读取的数据数目。对于我们的例子，队列中的数据读取完成后，toTake将为0，之后的事务不会再有实际的消息发送出去。第18行构建元数据，第19行调用emitPartitionBatch方法发送数据，该方法会在事务重传时被调用。

16.1.2 局部计数Bolt的实现

首先来看类BatchCount的实现，它用于完成局部计数，其代码如下：

```
1  public static class BatchCount extends BaseBatchBolt {
2      Object _id;
3      BatchOutputCollector _collector;
4  
5      int _count = 0;
6  
7      @Override
8      public void prepare(Map conf, TopologyContext context, BatchOutputCollector collector,
           Object id) {
9          _collector = collector;
10         _id = id;
11     }
12 
13     @Override
14     public void execute(Tuple tuple) {
15         _count++;
16     }
17 
18     @Override
19     public void finishBatch() {
20         _collector.emit(new Values(_id, _count));
```

```
21     }
22
23     @Override
24     public void declareOutputFields(OutputFieldsDeclarer declarer) {
25         declarer.declare(new Fields("id", "count"));
26     }
27 }
```

该类在execute方法中对_count变量进行自增操作。当属于一个事务的消息结束时，会调用finishBatch方法将该事务的局部计数结果发送出去。注意，declareOutputFields方法需要将事务序号作为第1列。

由于系统会为每个事务都创建一个新的BatchCount对象，故对于每个事务，_count都被初始化为0。

16.1.3　全局计数Bolt的实现

下面来看UpdateGlobalCount的实现，它实现了全局计数，其代码如下：

```
1  public static class UpdateGlobalCount extends BaseTransactionalBolt implementsICommitter {
2      TransactionAttempt _attempt;
3      BatchOutputCollector _collector;
4
5      int _sum = 0;
6
7      @Override
8      public void prepare(Map conf, TopologyContext context, BatchOutputCollector collector,
           TransactionAttempt attempt) {
9          _collector = collector;
10         _attempt = attempt;
11     }
12
13     @Override
14     public void execute(Tuple tuple) {
15         _sum+=tuple.getInteger(1);
16     }
17
18     @Override
19     public void finishBatch() {
20         Value val = DATABASE.get(GLOBAL_COUNT_KEY);
21         Value newval;
22         if(val == null || !val.txid.equals(_attempt.getTransactionId())) {
23             newval = new Value();
24             newval.txid = _attempt.getTransactionId();
25             if(val==null) {
26                 newval.count = _sum;
27            } else {
28                 newval.count = _sum + val.count;
29             }
30             DATABASE.put(GLOBAL_COUNT_KEY, newval);
31         } else {
```

```
32              newval = val;
33          }
34          _collector.emit(new Values(_attempt, newval.count));
35      }
36
37      @Override
38      public void declareOutputFields(OutputFieldsDeclarer declarer) {
39          declarer.declare(new Fields("id", "sum"));
40      }
41 }
```


- 第14~16行定义的execute方法会从输入消息的第1列获得局部计数结果，并累加至_sum变量中。注意，每次有新的事务时都会创建一个该对象，故_sum为一个事务内部的统计计数。
- 第19~35行定义的finishBatch方法模拟了去重操作。DATABASE用于存储全局计数结果。Value对象的键为事务序号，值为全局计数。若当前事务序号与Value中的事务序号相同，则当前正在处理的事务为事务重传，该事务对应的结果已经被更新至DATABASE对象中了，此时可以将该事务重传忽略。
- UpdateGlobalCount类继承自ICommitter，事务提交可以保证强序关系，故可以在此处去重。

16.2 构建 Topology

下面我们来看 Topology 的构建，相关代码如下：

```
1    public static final int PARTITION_TAKE_PER_BATCH = 3;
2    public static final Map<Integer, List<List<Object>>> DATA = new HashMap<Integer, List<List
         <Object>>>() {{
3        put(0, new ArrayList<List<Object>>() {{
4            add(new Values("cat"));
5            add(new Values("dog"));
6            add(new Values("chicken"));
7            add(new Values("cat"));
8            add(new Values("dog"));
9            add(new Values("apple"));
10       }});
11       put(1, new ArrayList<List<Object>>() {{
12           add(new Values("cat"));
13           add(new Values("dog"));
14           add(new Values("apple"));
15           add(new Values("banana"));
16       }});
17       put(2, new ArrayList<List<Object>>() {{
18           add(new Values("cat"));
19           add(new Values("cat"));
20           add(new Values("cat"));
21           add(new Values("cat"));
22           add(new Values("cat"));
23           add(new Values("dog"));
24           add(new Values("dog"));
```

```
25              add(new Values("dog"));
26              add(new Values("dog"));
27          }});
28      }};
29
30 MemoryTransactionalSpout spout = new MemoryTransactionalSpout(DATA, new Fields("word"), PARTITION_
       TAKE_PER_BATCH);
31 TransactionalTopologyBuilder builder = new TransactionalTopologyBuilder("global-count", "spout",
       spout, 3);
32 builder.setBolt("partial-count", new BatchCount(), 3)
33     .noneGrouping("spout");
34 builder.setBolt("sum", new UpdateGlobalCount())
35     .globalGrouping("partial-count");
```

在上述代码中，PARTITION_TAKE_PER_BATCH对应于每个事务中消息的数目，DATA用于模拟三个分区中的数据。

- 第30~31行定义并设置Spout节点，并行度为3，每个节点对应于一个分区。global-count为ZooKeeper元数据路径的一部分。
- 第32~33行定义局部计数节点，并行度为2。
- 第34~35行定义全局计数节点，它使用全局分组方式，并行度为1。

构建的 Topology 如图 16-1 所示，下面简要介绍一下。

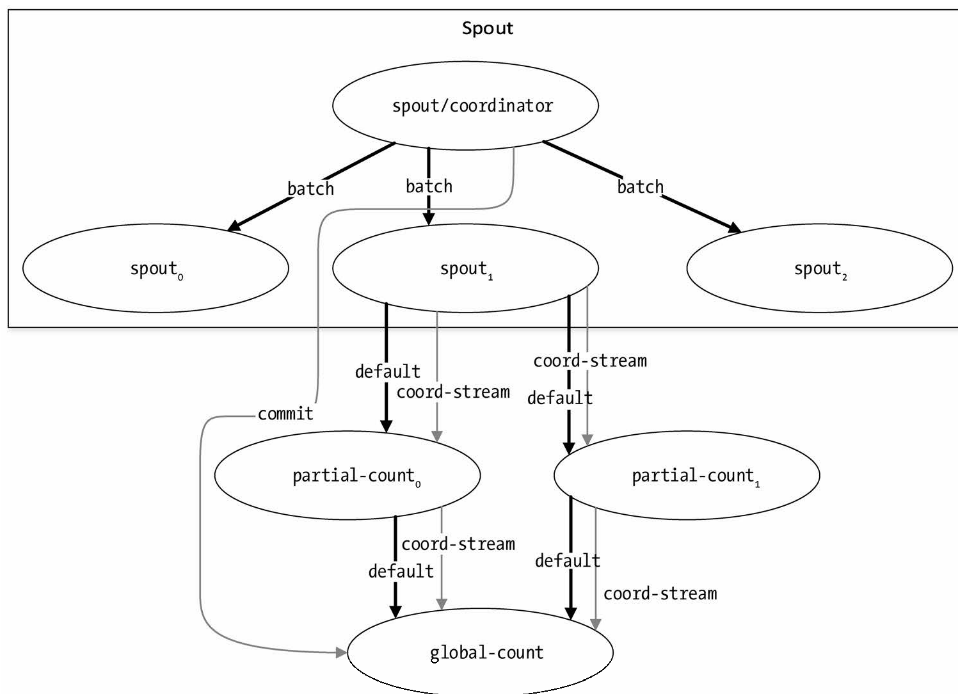


图16-1　事务Topology示意图

- 实线为数据消息，虚线为控制消息。
- 为了简化该图，图中只画了中间的$spout_1$节点（Emitter）与partial-count节点之间的流接收关系，$spout_0$和$spout_2$节点的流接收关系与中间的$spout_1$节点完全相同。
- spout/coordinator会向batch流发送事务的初始化消息。每个Spout节点对应该事务的一个分区。
- 每个spout节点会向default流发送数据消息，向coord-stream流发送协调消息。
- 每个partial-count节点会向global-count发送数据消息和协调消息。

16.3 事务处理示例

本节对事务1和事务2所发送的消息进行列表分析，理解这些消息的传送机制是理解整个系统的关键。

首先来看协调消息的参数设置，如表16-1所示。例如，$partial\text{-}count_0$节点所对应的表项表示其需要从三个源端Task接收协调消息。

表16-1 协调消息的参数设置

组　件	协调设置
spout\coordinator	无
$spout_0$	_sourceArgs：空 _idStreamSpec：空 _numSourceReports：0
$spout_1$	_sourceArgs：空 _idStreamSpec：空 _numSourceReports：0
$spout_2$	_sourceArgs：空 _idStreamSpec：空 _numSourceReports：0
$partial\text{-}count_0$	_sourceArgs：($spout_0$、$spout_1$、$spout_2$) _idStreamSpec：空 _numSourceReports：3
$partial\text{-}count_1$	_sourceArgs：($spout_0$、$spout_1$、$spout_2$) _idStreamSpec：空 _numSourceReports：3
global-count	_sourceArgs: ($partial\text{-}count_0$、$partial\text{-}count_1$) _idStreamSpec：spout/coordinator的commit流 _numSourceReports：2

表16-2与表16-3分别对应于两个事务的消息处理情况，其中包含协调消息，而具体的Ack消息则在此处被忽略。

表16-2　第一个事务发送消息示意

组　　件	发送的消息示意
spout\coordinator	(1, 消息ID)表示TxID=1
$spout_0$	(1, cat) 分区0对应的数据
	(1, dog) 分区0对应的数据
	(1, chicken) 分区0对应的数据
	(1, partial-$count_0$, 1) 协调消息，表示发送到partial-$count_0$ 一条消息
	(1, partial-$count_1$, 2) 协调消息，表示发送到partial-$count_1$ 一条消息
$spout_1$	(1, cat) 分区1的数据
	(1, dog)
	(1, apple)
	(1, partial-$count_0$, 1) 协调消息，表示发送到partial-$count_0$ 一条消息
	(1, partial-$count_1$, 2) 协调消息，表示发送到partial-$count_0$ 一条消息
$spout_2$	(1, cat) 分区2的数据
	(1, cat) 分区2的数据
	(1, cat) 分区2的数据
	(1, partial-$count_0$, 2) 协调消息，表示发送到partial-$count_0$ 两条消息
	(1, partial-$count_1$, 1) 协调消息，表示发送到partial-$count_1$ 一条消息
partial-$count_0$	(1, 4) 可能有4条消息到达该节点
	(1, global-count, 4) 协调消息，向global-count节点发送一条消息
partial-$count_1$	(1, 5) 另外5条消息到达该节点
	(1, global-count, 5) 协调消息，向global-count节点发送一条消息
global-count	(1, 9)

表16-3　第二个事务发送消息示意

组　　件	发送的消息示意
spout\coordinator	(2, 消息ID)表示TxID=2
$spout_0$	(2, dog) 分区0对应的数据
	(2,apple) 分区0对应的数据
	(2, partial-$count_0$, 1) 协调消息，表示发送到partial-$count_0$ 一条消息
	(2, partial-$count_1$, 1) 协调消息，表示发送到partial-$count_1$ 一条消息
$spout_1$	(2, banana) 分区1的数据
	(2, partial-$count_0$, 1) 协调消息，表示发送到partial-$count_0$ 一条消息
	(2, partial-$count_1$, 0) 协调消息，表示发送到partial-$count_1$零条消息
$spout_2$	(2, cat) 分区2的数据
	(2, cat) 分区2的数据
	(2, dog) 分区2的数据
	(2, partial-$count_0$, 2) 协调消息，表示发送到partial-$count_0$ 两条消息
	(2, partial-$count_1$, 1) 协调消息，表示发送到partial-$count_1$ 一条消息

（续）

组　　件	发送的消息示意
partial-count$_0$	(1, 2) 可能有两条消息到达该节点
	(1, global-count, 2) 协调消息，向global-count节点发送一条消息
partial-count$_1$	(1,3) 另外3条消息到达该节点
	(2, global-count, 3) 协调消息，向global-count节点发送一条消息
global-count	(1, 9)
	(2, 5)

在事务Topology中，系统只会跟踪控制消息，而不会跟踪数据消息。控制消息包括协调Spout发送给消息发送Bolt的事务消息，以及Bolt之间的协调消息等，这些消息的根为事务消息。

表16-4以Acker Bolt收到的消息为中心分析事务Topology的消息跟踪过程。为了简化起见认为每个节点只发送一条消息。

表16-4　事务Topology的消息跟踪

源组件	Acker Bolt收到的消息示意
spout\coordinator	(TxIdRootId, AckValue，_ack_init)
spout$_0$	(TxIdRootId, AckValue，_ack_ack)
spout$_1$	(TxIdRootId, AckValue，_ack_ack)
spout$_2$	(TxIdRootId, AckValue，_ack_ack)
partial-count$_0$	(TxIdRootId, AckValue，_ack_ack)
partial-count$_1$	(TxIdRootId, AckValue，_ack_ack)
global-count	(TxIdRootId, AckValue，_ack_ack)
	此时该事务所对应的跟踪值为零，Acker Bolt将向spout\coordinator发送消息，表示该事务的处理阶段结束。
	在事务提交阶段，消息跟踪则较为简单

注意，AckValue是根据发送出去的控制消息来计算得到的。在事务的处理阶段，Acker Bolt一共收到7条消息，分别对应于跟踪的初始化和跟踪值的更新。

第 17 章

Trident的Spout节点

17

从本章开始，我们将介绍Trident的实现。Trident是Storm提供给用户的另外一套接口，它提供了基本的流处理功能以及可靠的消息处理功能，其中对流的操作是Trident的核心。读者需要首先阅读下面链接的内容以获得对Trident的宏观认知，然后参照本书接下来各章的思路学习其实现。

- Trident的例子：https://github.com/nathanmarz/storm/wiki/Trident-tutorial。
- Trident的API接口概述：https://github.com/nathanmarz/storm/wiki/Trident-API-Overview。
- Trident的Spout类型：https://github.com/nathanmarz/storm/wiki/Trident-spouts。
- Trident的存储抽象：https://github.com/nathanmarz/storm/wiki/Trident-state。

Trident的实现较为复杂，本书将分成多个章节对其进行讨论，本章首先介绍Trident中的Spout节点类型。

Trident主要支持两种类型的Spout节点：`ITridentSpout`以及DRPC Spout。对于Storm中其他基本类型的Spout，例如`IRichSpout`和`IBatchSpout`，Trident进行了接口适配，将它们适配成为`ITridentSpout`接口并在Topology中执行。主要的类关系如图17-1所示。

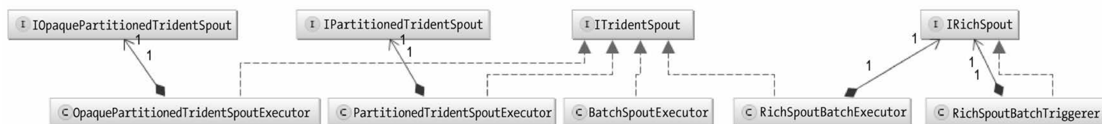


图17-1　Trident中Spout节点的类关系

本章将详细分析这些Spout节点及适配Spout节点的执行器。

17.1　ITridentSpout 接口

`ITridentSpout`是Trident中唯一支持的Spout类型。与`ITransactionalSpout`接口类似，`ITridentSpout`接口主要分成两个部分：一部分为协调Spout接口，另外一部分为消息发送Bolt接口。不同于事务Topology中的协调Spout节点，在Trident里协调Spout的逻辑会被部署到多个节点中运行，而消息发送节点则会被部署到Bolt中执行。下面首先介绍其协调Spout接口。

17.1.1 BatchCoordinator 接口

BatchCoordinator接口对应于批协调Spout接口（简称协调Spout），顾名思义，Trident中的处理主要是基于批处理的。该接口的代码如下：

```
1 public interface BatchCoordinator<X> {
2     /**
3      * Create metadata for this particular transaction id which has never
4      * been emitted before. The metadata should contain whatever is necessary
5      * to be able to replay the exact batch for the transaction at a later point.
6      *
7      * The metadata is stored in Zookeeper.
8      *
9      * Storm uses the Kryo serializations configured in the component configuration
10     * for this spout to serialize and deserialize the metadata.
11     *
12     * @param txid The id of the transaction.
13     * @param prevMetadata The metadata of the previous transaction
14     * @param currMetadata The metadata for this transaction the last time it wasinitialized.
15     *                     null if this is the first attempt
16     * @return the metadata for this new transaction
17     */
18    X initializeTransaction(long txid, X prevMetadata, X currMetadata);
19
20    void success(long txid);
21
22    boolean isReady(long txid);
23
24    /**
25     * Release any resources from this coordinator.
26     */
27     void close();
28 }
```

- initializeTransaction方法返回通用类型的变量X，它是事务的元数据。X是用户自定义的与事务相关的数据类型，返回的数据会被存储到ZooKeeper中。
  在initializeTransaction方法中，txid为事务序号，prevMetadata为前一个事务所对应的元数据。若当前事务为第一个事务，则prevMetadata为空。currMetadata是当前事务的元数据，如果是当前事务的第一次尝试，则为空，否则为事务第一次尝试时所产生的元数据。
- isReady方法用来判断事务所对应的数据是否已经准备好了，当它返回true时，表示可以开始一个新事务。在Trident中，isReady方法传入了当前的事务序号，这是对传统事务Topology的一个改进。基于传入的事务序号， isReady可以获取相关的元数据。

BatchCoordinator中实现的方法会被部署到多个节点中运行，其中isReady是在真正的Spout（MasterBatchCoordinator）中执行的，而其他方法在TridentSpoutCoordinator中执行。理解这些方法的调用时机及场景对于实现接口是很有帮助的。

17.1.2　TridentSpoutCoordinator

TridentSpoutCoordinator类为BatchCoordinator的执行器，它继承自IBasicBolt接口，实质上为Bolt节点，主要用于执行协调Spout接口中的initializeTransaction方法。可以看出，事务的元数据是在Bolt节点中产生的。该类的代码和分析如下：

```
1 public class TridentSpoutCoordinator implements IBasicBolt {
2     public static final Logger LOG = LoggerFactory.getLogger(TridentSpoutCoordinator.class);
3     private static final String META_DIR = "meta";
4
5     ITridentSpout _spout;
6     ITridentSpout.BatchCoordinator _coord;
7     RotatingTransactionalState _state;
8     TransactionalState _underlyingState;
9     String _id;
10
11    public TridentSpoutCoordinator(String id, ITridentSpout spout) {
12        _spout = spout;
13        _id = id;
14    }
15
16    @Override
17    public void prepare(Map conf, TopologyContext context) {
18        _coord = _spout.getCoordinator(_id, conf, context);
19        _underlyingState = TransactionalState.newCoordinatorState(conf, _id);
20        _state = new RotatingTransactionalState(_underlyingState, META_DIR);
21    }
22
23    @Override
24    public void execute(Tuple tuple, BasicOutputCollector collector) {
25        TransactionAttempt attempt = (TransactionAttempt) tuple.getValue(0);
26
27        if(tuple.getSourceStreamId().equals(MasterBatchCoordinator.SUCCESS_STREAM_ID)) {
28            _state.cleanupBefore(attempt.getTransactionId());
29            _coord.success(attempt.getTransactionId());
30        } else {
31            long txid = attempt.getTransactionId();
32            Object prevMeta = _state.getPreviousState(txid);
33            Object meta = _coord.initializeTransaction(txid, prevMeta, _state.getState(txid));
34            _state.overrideState(txid, meta);
35            collector.emit(MasterBatchCoordinator.BATCH_STREAM_ID, new Values(attempt, meta));
36        }
37
38    }
39
40    @Override
41    public void cleanup() {
42        _coord.close();
43        _underlyingState.close();
44    }
45
46    @Override
```

```
47     public void declareOutputFields(OutputFieldsDeclarer declarer) {
48         declarer.declareStream(MasterBatchCoordinator.BATCH_STREAM_ID, new Fields("tx",
               "metadata"));
49     }
50 
51     @Override
52     public Map<String, Object> getComponentConfiguration() {
53         Config ret = new Config();
54         ret.setMaxTaskParallelism(1);
55         return ret;
56     }
57 }
```


- _spout为ITridentSpout的对象引用。这里主要通过调用_spout的getCoordinator方法来获得BatchCoordinator的引用，然后将该引用存储于_coord成员变量中。_state和_underlyingState用来维护ZooKeeper中的元数据。
- 在execute方法的实现中，TridentSpoutCoordinator将接收$success和$batch 流，这两个流只包含一列，即事务序号txid。
  - 当收到$success流的消息时，表明事务处理已经结束，于是调用_coord的success方法，同时清理ZooKeeper中的元数据。
  - 当收到$batch消息时，execute方法会初始化一个事务并将消息发送到$batch流中，消息的格式为<tx，metadata>。由于收到的消息可能是重传的消息，例如由Spout的超时引发的重传，此时事务所对应的元数据可能已经存在，这也是initializeTransaction方法有prevMeta参数的原因。这与事务Topology是不同的，后者对于一个事务，其initializeTransaction方法只会被调用一次。
  - 注意到Trident是在Bolt节点中对事务进行初始化的，这与事务Topology也不同，后者在Spout节点中产生。Trident中的Spout节点更类似于一个脉冲服务器。
- 第52~56行规定了TridentSpoutCoordinator的并行度为1。

17.1.3 MasterBatchCoordinator

MasterBatchCoordinator类是Trident中真正的Spout节点。Trident中可以含有多个MasterBatchCoordinator类型的Spout节点，每个Spout节点又进一步对应于一个含有存储节点的节点组（关于节点组的相关内容，可参阅24.2.1节）。

每个MasterBatchCoordinator节点可以对应多个ITridentSpout节点，这些ITridentSpout节点属于同一个节点组。

MasterBatchCoordinator类用来产生一个新的事务以及判断一个事务是否已经被成功处理。下面首先分析其成员变量。该类的定义如下：

```
public class MasterBatchCoordinator extends BaseRichSpout {
    public static final Logger LOG =LoggerFactory.getLogger(MasterBatchCoordinator.class);

    public static final long INIT_TXID = 1L;
```

```
public static final String BATCH_STREAM_ID = "$batch";
public static final String COMMIT_STREAM_ID = "$commit";
public static final String SUCCESS_STREAM_ID = "$success";

private static final String CURRENT_TX = "currtx";
private static final String CURRENT_ATTEMPTS = "currattempts";

private List<TransactionalState> _states = new ArrayList();

TreeMap<Long, TransactionStatus> _activeTx = new TreeMap<Long,TransactionStatus>();
TreeMap<Long, Integer> _attemptIds;

private SpoutOutputCollector _collector;
Long _currTransaction;
int _maxTransactionActive;

List<ITridentSpout.BatchCoordinator> _coordinators = new ArrayList();

List<String> _managedSpoutIds;
List<ITridentSpout> _spouts;
WindowedTimeThrottler _throttler;

boolean _active = true;
```

- INIT_TXID表示最开始的事务序号，它从1开始，为长整型。
- $batch、$commit和$success为Spout产生的3种流，流中的消息对应于事务的某个特定阶段。
  - $batch对应于事务的处理阶段（PROCESSING）。
  - $commit对应于事务处理结束阶段（PROCESSED），表示对事务中消息的处理已经完成，事务等待被提交。
  - $success对应于提交阶段（COMMITTING），表示事务已经处理结束，通过向该流发送消息以通知相关节点。相关节点收到该消息后，可以进行事务提交处理，也可以进行历史事务数据的清理工作。

  系统会对发送到$batch和$commit的消息进行跟踪，而不会对发送到$success的消息进行跟踪。
- currtx以及currattempts对应于ZooKeeper中的元数据路径，分别用来存储当前的事务和事务所对应的尝试。Trident中事务尝试号不再采用长整型的随机数，而是采用递增的整数。
- _spouts以及_managedSpoutIds用来存储MasterBatchCoordinator所管理的Spout节点。在Trident中，同一个节点组可以包含多个ITridentSpout类型的Spout节点，并集中利用MasterBatchCoordinator来进行管理。
- _states用来存储每个ITridentSpout所对应的元数据。
- _activeTx用来保存每一个事务的尝试状态。
- _attemptIds用来存储每一个事务当前的尝试编号。
- _currTransaction表示下一个需要进行提交的事务。
- _maxTransactionActive表示同时运行的事务数目的最大值。

- _coordinators用来存储ITridentSpout中的BatchCoordinator。

接下来对该类的主要函数进行分析。open函数主要用来对_states对象进行初始化，该函数会根据ZooKeeper所保存的内容进行数据恢复，以便于系统从上次结束的地方继续运行，其代码如下：

```
1 public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
2     _throttler = newWindowedTimeThrottler((Number)conf.get(Config.TOPOLOGY_TRIDENT_BATCH_
          EMIT_INTERVAL_MILLIS), 1);
3     for(String spoutId: _managedSpoutIds) {
4         _states.add(TransactionalState.newCoordinatorState(conf, spoutId));
5     }
6     _currTransaction = getStoredCurrTransaction();
7
8     _collector = collector;
9     Number active = (Number) conf.get(Config.TOPOLOGY_MAX_SPOUT_PENDING);
10    if(active==null) {
11        _maxTransactionActive = 1;
12    } else {
13        _maxTransactionActive = active.intValue();
14    }
15    _attemptIds = getStoredCurrAttempts(_currTransaction, _maxTransactionActive);
16
17
18    for(int i=0; i<_spouts.size(); i++) {
19        String txId = _managedSpoutIds.get(i);
20        _coordinators.add(_spouts.get(i).getCoordinator(txId, conf, context));
21    }
22 }
```

- 第6行获取正在运行的事务序号。这里会以所有Spout中存储的最大的事务序号作为_currTransaction的值。

  目前这个类中管理的每一个Spout节点都需要独立地维护其当前事务及每个事务的当前尝试序号。不过在目前的实现中，每个Spout的这些数据都是一致的，这增加了额外的同步负担，我认为这是需要改进的地方。
- 第15行获取每个事务所对应的当前尝试值。与前面类似，这里也会取所有Spout中存储的关于_currTransaction事务序号中的最大值。
- 第18~21行初始化_coordinators，即该Spout代理的ITridentSpout节点。

declareOutputFields方法定义输出的三个流，它们的模式均为只包含要处理的事务序号。理解这些流是如何被接收的，对于理解整个系统来说十分重要。该方法的代码如下：

```
public void declareOutputFields(OutputFieldsDeclarer declarer) {
    // in partitioned example, in case an emitter task receives a later transaction
          than it's emitted so far,
    // when it sees the earlier txid it should know to emit nothing
    declarer.declareStream(BATCH_STREAM_ID, new Fields("tx"));
    declarer.declareStream(COMMIT_STREAM_ID, new Fields("tx"));
    declarer.declareStream(SUCCESS_STREAM_ID, new Fields("tx"));
}
```

Sync方法主要用于产生新的事务，其代码如下：

```
1 private void sync() {
2    // note that sometimes the tuples active may be less than max_spout_pending, e.g.
3    // max_spout_pending = 3
4    // tx 1, 2, 3 active, tx 2 is acked. there won't be a commit for tx 2 (because
          tx 1 isn't committed yet),
5    // and there won't be a batch for tx 4 because there's max_spout_pending tx active
6    TransactionStatus maybeCommit = _activeTx.get(_currTransaction);
7    if(maybeCommit!=null && maybeCommit.status == AttemptStatus.PROCESSED) {
8        maybeCommit.status = AttemptStatus.COMMITTING;
9        _collector.emit(COMMIT_STREAM_ID, new Values(maybeCommit.attempt), maybeCommit.attempt);
10   }
11
12   if(_active) {
13       if(_activeTx.size() < _maxTransactionActive) {
14           Long curr = _currTransaction;
15           for(int i=0; i<_maxTransactionActive; i++) {
16               if(!_activeTx.containsKey(curr) && isReady(curr)) {
17               // by using a monotonically increasing attempt id, downstream tasks
18               // can be memory efficient by clearing out state for old attempts
19               // as soon as they see a higher attempt id for a transaction
20               Integer attemptId = _attemptIds.get(curr);
21               if(attemptId==null) {
22                   attemptId = 0;
23               } else {
24                   attemptId++;
25               }
26               _attemptIds.put(curr, attemptId);
27               for(TransactionalState state: _states) {
28                   state.setData(CURRENT_ATTEMPTS, _attemptIds);
29               }
30
31               TransactionAttempt attempt = new TransactionAttempt(curr, attemptId);
32               _activeTx.put(curr, new TransactionStatus(attempt));
33               _collector.emit(BATCH_STREAM_ID, new Values(attempt), attempt);
34               _throttler.markEvent();
35           }
36           curr = nextTransactionId(curr);
37            }
38       }
39    }
40 }
```

- 第7~10行判断当前事务_currTransaction的状态是否为处理成功阶段，若已被成功处理，则向$commit流发送消息，收听该流的节点会根据这个消息来调用finishBatch方法，从而完成对事务的提交和后处理。
- 第12~38行用来判断是否产生一个新的事务。系统中最多允许_maxTransactionActive个事务同时运行，当前活跃的事务的序号区间为 [_currTxId, _currTxId+_maxTransactionActive-1]。注意，只有在当前事务结束之后，系统才会初始化新的事务，所以系统中实

际活跃的事务可能少于_maxTransactionActive中定义的数目。

- 第16行根据isReady方法来判断是否可以初始化一个新事务。目前，只要任何一个BatchCoordinator的isReady方法返回true，系统就会开始一个新事务。这部分是需要改进的。例如，两个流要进行连接时，需要保证在对应的两个Spout中，消息发送节点所发送的数据处于同一个事务序号下，这样才能连接，这也要求isReady方法应该同时返回true。下面的代码为目前isReady的实现：

```
private boolean isReady(long txid) {
    if(_throttler.isThrottled()) return false;
    //TODO: make this strategy configurable?... right now it goes if anyone is ready
    for(ITridentSpout.BatchCoordinator coord: _coordinators) {
        if(coord.isReady(txid)) return true;
    }
    return false;
}
```

- 第33行向$batch流发送新初始化的事务消息。注意，所有发送出去的消息都会被跟踪。其他节点则通过全局分组的方式进行收听。

当发送至$batch以及$commit的消息被Ack后，下面的方法就会被调用：

```
1  public void ack(Object msgId) {
2      TransactionAttempt tx = (TransactionAttempt) msgId;
3      TransactionStatus status = _activeTx.get(tx.getTransactionId());
4      if(status!=null && tx.equals(status.attempt)) {
5          if(status.status==AttemptStatus.PROCESSING) {
6              status.status = AttemptStatus.PROCESSED;
7          } else if(status.status==AttemptStatus.COMMITTING) {
8              _activeTx.remove(tx.getTransactionId());
9              _attemptIds.remove(tx.getTransactionId());
10             _collector.emit(SUCCESS_STREAM_ID, new Values(tx));
11             _currTransaction = nextTransactionId(tx.getTransactionId());
12             for(TransactionalState state: _states) {
13                 state.setData(CURRENT_TX, _currTransaction);
14             }
15         }
16         sync();
17     }
18 }
```

- 第5~7行收到发送至$batch流的Ack消息，表明事务的数据处理已经结束（事务中的数据在Bolt节点上均已调用execute方法），此时将事务的状态改为PROCESSED。
- 第8~14行收到发送至$commit的Ack消息，表示该事务提交已经被成功处理。此时会向$success流发送消息，相关节点根据该消息进行清理及后处理等操作。第11行将当前事务序号更新为下一个事务序号。第12~14行将每一个Spout所对应的当前事务信息更新到ZooKeeper中。

在收到失败消息时，失败的事务及其后续事务都需要重传，但事务的元数据并不会重新产生，

而是利用之前初始化的内容获得。fail方法的实现代码如下：

```
1 public void fail(Object msgId) {
2     TransactionAttempt tx = (TransactionAttempt) msgId;
3     TransactionStatus stored = _activeTx.remove(tx.getTransactionId());
4     if(stored!=null && tx.equals(stored.attempt)) {
5         _activeTx.tailMap(tx.getTransactionId()).clear();
6         sync();
7     }
8 }
```

17.1.4　消息发送节点接口

ITridentSpout的消息发送节点接口逻辑会被部署到Bolt节点中运行。消息发送节点会收听协调Spout的$batch和$success流。当收到源自$batch流中的消息时，节点便调用emitBatch方法来发送数据。消息发送节点的接口定义如下：

```
public interface Emitter<X> {
    /**
     * Emit a batch for the specified transaction attempt and metadata for the transaction. The metadata
     * was created by the Coordinator in the initializeTranaction method. This method must always emit
     * the same batch of tuples across all tasks for the same transaction id.
     *
     */
    void emitBatch(TransactionAttempt tx, X coordinatorMeta, TridentCollector collector);

    /**
     * This attempt committed successfully, so all state for this commit and before
         can be safely cleaned up.
     */
    void success(TransactionAttempt tx);

    /**
     * Release any resources held by this emitter.
     */
    void close();
}
```

- X类型的coordinatorMeta由BatchCoordinator中的initializeTransaction方法初始化得到。
- 参数collector用来向外发送数据。由于Trident可能将多个操作放在一个Bolt中执行，此处的collector可能只是调用同一个Bolt中的其他操作来处理其输出，而并不是真正向外发送数据，详情可参见第24章。
- 当收到$success 流的消息时，会调用success方法对事务进行后处理等。

17.1.5　消息发送接口的执行器

ITridentSpout中的消息发送接口逻辑是在TridentSpoutExecutor中执行的，该类实现了

ITridentBatchBolt接口。TridentSpoutExecutor会在其execute方法中根据收到消息的流号来调用消息发送接口中的相应方法。这里主要分析其execute方法的实现：

```
1 public void execute(BatchInfo info, Tuple input) {
2     // there won't be a BatchInfo for the success stream
3     TransactionAttempt attempt = (TransactionAttempt) input.getValue(0);
4     if(input.getSourceStreamId().equals(MasterBatchCoordinator.COMMIT_STREAM_ID)) {
5         if(attempt.equals(_activeBatches.get(attempt.getTransactionId()))) {
6             ((ICommitterTridentSpout.Emitter) _emitter).commit(attempt);
7             _activeBatches.remove(attempt.getTransactionId());
8         } else {
9              throw new FailedException("Received commit for different transactionattempt");
10        }
11    } else if(input.getSourceStreamId().equals(MasterBatchCoordinator.SUCCESS_STREAM_ID)) {
12        // valid to delete before what's been committed since
13        // those batches will never be accessed again
14        _activeBatches.headMap(attempt.getTransactionId()).clear();
15        _emitter.success(attempt);
16    } else {
17        _collector.setBatch(info.batchId);
18        _emitter.emitBatch(attempt, input.getValue(1), _collector);
19        _activeBatches.put(attempt.getTransactionId(), attempt);
20    }
21 }
```

- 通常，BatchInfo对象中含有与输入消息input的第1列相同的元素。不过对于$success流，传入的BatchInfo对象为空，相关内容会在第26章中进一步讨论。第3行统一地从输入消息地第1列获得事务信息。
- 在第4~10行中，只有在实现了ICommitterTridentSpout接口后，Topology构建器才会去收听$commit流。收到从这个流发来的消息时execute方法将调用消息发送接口Emitter的commit方法。
- 第12~15行处理来自$success流的消息，这里调用_emitter的success方法。
- 第16~21行处理来自$batch流的消息，这里调用_emitter的emitBatch方法发送消息，其中_collector为AdIdCollector类型。AdIdCollector在发送消息时，会将事务序号添加到第1列。
- _activeBatches用来存储当前节点上运行的事务以及尝试编号，以便在实现了ICommitterTridentSpout的节点上保证提交的事务编号与$commit流消息中的事务序号相同。它主要用于模糊事务类型，即同一个事务的不同尝试都可能对应着不同的数据，事务提交时必须保证使用了正确的事务以及尝试编号，仅仅事务编号相同是不够的。
- 第14行清除掉当前事务编号之前的事务数据，而第7行则清除掉了当前的事务。

17.2 适配 IRichSpout 接口

为了更好地与其他接口相兼容，Trident对基础的Spout接口进行了适配。

在Trident中，IRichSpout并不是放在Spout节点中运行，而是在Bolt节点中。默认的Spout节点

会处理消息超时，以及对Ack、Fail方法的回调等，然而通常的Bolt节点并不具备这些功能。

RichSpoutBatchExecutor类对这些行为进行了适配模拟，IRichSpout接口被适配成为ITridentSpout接口。

Spout节点通过ISpoutOutputCollector类来发送消息，这与Bolt节点中使用OutputCollector是不一致的，因此需要类CaptureCollector对其进行适配。下面首先分析CaptureCollector，其代码如下：

```
static class CaptureCollector implements ISpoutOutputCollector {

    TridentCollector _collector;
    public List<Object> ids;
    public int numEmitted;

    public void reset(TridentCollector c) {
        _collector = c;
        ids = new ArrayList<Object>();
    }

    @Override
    public void reportError(Throwable t) {
        _collector.reportError(t);
    }

    @Override
    public List<Integer> emit(String stream, List<Object> values, Object id) {
        if(id!=null) ids.add(id);
        numEmitted++;
        _collector.emit(values);
        return null;
    }

    @Override
    public void emitDirect(int task, String stream, List<Object> values, Object id) {
        throw new UnsupportedOperationException("Trident does not support direct streams");
    }
}
```

- _collector为TridentCollector类型，其实际类型为Bolt的OutputCollector。
- _ids成员变量负责记录用于消息跟踪的消息号MessageId。
- numEmitted用来记录发送出去的消息条数，以此实现流量控制。Spout通过maxSpoutPending来完成控制，当发送消息的数目过多时，Spout节点将处于不活跃状态。

接下来分析消息发送接口的实现，相关代码如下：

```
1 class RichSpoutEmitter implements ITridentSpout.Emitter<Object> {
2     int _maxBatchSize;
3     boolean prepared = false;
4     CaptureCollector _collector;
5     RotatingMap<Long, List<Object>> idsMap;
```

```
6     Map _conf;
7     TopologyContext _context;
8     long lastRotate = System.currentTimeMillis();
9     long rotateTime;
10
11    public RichSpoutEmitter(Map conf, TopologyContext context) {
12        _conf = conf;
13        _context = context;
14        Number batchSize = (Number) conf.get(MAX_BATCH_SIZE_CONF);
15        if(batchSize==null) batchSize = 1000;
16        _maxBatchSize = batchSize.intValue();
17        _collector = new CaptureCollector();
18        idsMap = new RotatingMap(3);
19        rotateTime = 1000L * ((Number)conf.get(Config.TOPOLOGY_MESSAGE_TIMEOUT_SECS))
              .intValue();
20 }
21
22 @Override
23     public void emitBatch(TransactionAttempt tx, Object coordinatorMeta, TridentCollector
          collector) {
24     long txid = tx.getTransactionId();
25
26     long now = System.currentTimeMillis();
27     if(now - lastRotate > rotateTime) {
28         Map<Long, List<Object>> failed = idsMap.rotate();
29         for(Long id: failed.keySet()) {
30             //TODO: this isn't right... it's not in the map anymore
31             fail(id);
32         }
33         lastRotate = now;
34     }
35
36     if(idsMap.containsKey(txid)) {
37         fail(txid);
38     }
39
40     _collector.reset(collector);
41     if(!prepared) {
42         _spout.open(_conf, _context, new SpoutOutputCollector(_collector));
43         prepared = true;
44     }
45     for(int i=0; i<_maxBatchSize; i++) {
46         _spout.nextTuple();
47         if(_collector.numEmitted < i) {
48             break;
49         }
50     }
51     idsMap.put(txid, _collector.ids);
52
53     }
54
55     @Override
56     public void success(TransactionAttempt tx) {
57         ack(tx.getTransactionId());
```

```
58      }
59
60      private void ack(long batchId) {
61          List<Object> ids = (List<Object>) idsMap.remove(batchId);
62          if(ids!=null) {
63              for(Object id: ids) {
64                  _spout.ack(id);
65              }
66          }
67      }
68
69      private void fail(long batchId) {
70          List<Object> ids = (List<Object>) idsMap.remove(batchId);
71          if(ids!=null) {
72              for(Object id: ids) {
73                  _spout.fail(id);
74              }
75          }
76      }
77  }
```

- _maxBatchSize的作用类似于maxSpoutPending变量，它根据MAX_BATCH_SIZE_CONF来设置。
- rotateTime由TOPOLOGY_MESSAGE_TIMEOUT_SECS设置，用来控制消息超时。
- idsMap用来保存Spout在发送消息时所附带的消息序号MessageId。键为事务序号，值为属于该事务的所有消息的消息号。
- 第23~53行实现了emitBatch方法。
  - 第27~34行根据超时时间设置来判断是否该对消息进行失败处理，这里将调用Spout的fail方法。
  - 第36~38行对当前事务所属的消息调用fail方法，主要来应对事务重传的情况。
  - 第40行对_collector调用reset方法，清空已缓存的消息序号MessageId。
  - 第41~44行调用Spout的open方法，并设置prepared变量来防止多次调用。
  - 第45~50行不断调用Spout的nextTuple方法来产生消息。注意第42行传入了利用CaptureCollector初始化的SpoutOutputCollector。所以Spout实际上是通过Bolt的OutputCollector发送消息的，并且对消息号MessageId进行了记录。当一次nextTuple函数调用未向外发送任何消息时，称为一次空跑。此处对空跑的控制是有问题的，由于_collector的numEmitted变量并没有被重置，这将导致接下来的事务对空跑的控制是失效的。故需要在CaptureCollector的reset方法中将numEmitted设置为0。
- 在第56~58行中，收到$success流的消息时，表明一个事务的处理已经完成，于是调用_spout的Ack方法。这里我们看到了$success是如何被利用的。

ITridentSpout的Coordinator接口实现起来很简单，它只是在isReady方法中返回true。

17.3 适配 IBatchSpout 接口

IBatchSpout和ITridentSpout中的消息发送接口较为接近，都是进行批处理的。在Trident中，我们利用BatchSpoutExecutor来执行IBatchSpout。类似地，BatchSpoutExecutor也实现了ITridentSpout接口，下面来看其接口实现：

```
public class BatchSpoutEmitter implements Emitter {

    @Override
    public void emitBatch(TransactionAttempt tx, Object coordinatorMeta,TridentCollector collector) {
       _spout.emitBatch(tx.getTransactionId(), collector);
    }

    @Override
    public void success(TransactionAttempt tx) {
        _spout.ack(tx.getTransactionId());
    }

    @Override
    public void close() {
        _spout.close();
    }
}
```

- 在emitBatch方法中，我们调用了_spout的emitBatch方法。IBatchBolt不需要元数据支持。在success方法中，我们调用了_spout的ack方法，表示一个事务已经处理结束。
- Coordinator的实现只是在isReady方法中返回true，同样地，事务相关的元数据为空。
- IBatchSpout的emitBatch方法需要传入事务序号。实现IBatchSpout的用户类可以利用事务序号与元数据进行关联。

17.4 Trident 中分区的 Spout 类型

Trident提供了类似于分区事务Topology的接口类型，采用的实现技术也是相似的，即首先定义用户接口，然后定义一个执行类，并将该接口适配成为ITridentSpout接口。本节将对Trident中的分区Spout类型进行分析。

17.4.1 分区 Spout 接口

IPartitionedTridentSpout接口用于处理输入数据已经被分区的情况。例如在Kafka队列中，数据以分区的形式存放在不同的机器（Broker）上，此时便可实现这个接口来完成对Kafka的数据读取，每个事务都会分别从一个分区上读取一定的数据。

事务的元数据存储于ZooKeeper中。当事务处理失败后，会利用该元数据重新读取Kafka队列，这样可以保证事务重传时处理的是相同的数据。

IPartitionedTridentSpout接口与IPartitionedTransactionalSpout接口是很类似的，读者可以对比这两个接口的异同来学习。IPartitionedTransactionalSpout的代码如下：

```
/**
 * This interface defines a transactional spout that reads its tuples from a partitioned set of
 * brokers. It automates the storing of metadata for each partition to ensure that the same batch
 * is always emitted for the same transaction id. The partition metadata is stored in Zookeeper.
 */
public interface IPartitionedTridentSpout<Partitions, Partition extends ISpoutPartition, T> extends
    Serializable {
    public interface Coordinator<Partitions> {
        /**
         * Return the partitions currently in the source of data. The idea is
         * is that if a new partition is added and a prior transaction is replayed, it doesn't
         * emit tuples for the new partition because it knows what partitions were in
         * that transaction.
         */
        Partitions getPartitionsForBatch();

        boolean isReady(long txid);

        void close();
    }

    public interface Emitter<Partitions, Partition extends ISpoutPartition, X> {

        List<Partition> getOrderedPartitions(Partitions allPartitionInfo);

        /**
         * Emit a batch of tuples for a partition/transaction that's never been emitted before.
         * Return the metadata that can be used to reconstruct this partition/batch in the future.
         */
        X emitPartitionBatchNew(TransactionAttempt tx, TridentCollector collector, Partition
             partition, X lastPartitionMeta);

        /**
         * This method is called when this task is responsible for a new set of partitions. Should be
              used
         * to manage things like connections to brokers.
         */
        void refreshPartitions(List<Partition> partitionResponsibilities);

        /**
         * Emit a batch of tuples for a partition/transaction that has been emitted before, using
         * the metadata created when it was first emitted.
         */
        void emitPartitionBatch(TransactionAttempt tx, TridentCollector collector, Partition
           partition, X partitionMeta);
        void close();
    }

    Coordinator<Partitions> getCoordinator(Map conf, TopologyContext context);
    Emitter<Partitions, Partition, T> getEmitter(Map conf, TopologyContext context);
```

```
    Map getComponentConfiguration();
    Fields getOutputFields();
}
```

- IPartitionedTridentSpout中有3个通用类型：Partitions、Partition和X。Coordinator中存储数据的类型为Partitions，为通用类型，其含义为分区的元数据。例如，一共存在多少个分区，分区所在的Broker，等等都属于这种类型。具体信息由用户定义，不过这部分数据是需要保持相对稳定的。
- 在IPartitionedTrident接口中，通过调用getOrderedPartitions方法根据输入的Partitions类型获得Partition的列表。同样地，Partition也为通用类型，但是它要继承自ISpoutPartition接口，即含有getId方法来获得Partition的编号ID。
- 类型X表示处理某个Partition在某一个事务上面对应的元数据类型。例如，这个事务要读取的Partition数据的区间。
- Coordinator的isReady方法用来判断输入的事务是否可以开始，例如，可以判断Kafka队列中是否存在新数据。
- Emitter的getOrderedPartitions方法会在分区元数据发生变化（即Partitions发生变化）时被调用。该方法与refreshPartitions的调用时机相同，用来应对分区的变化。例如，建立并维护与新增加Partition的连接就可以使用这个方法。
- emitPartitionBatchNew方法会在事务第一次尝试时被调用，其参数lastPartitionMeta表示该分区的上一次事务所对应的元数据。该方法将返回当前事务在当前分区的元数据，该数据会被存储在ZooKeeper中。
- emitPartitionBatch方法会在事务被重试时调用，其元数据是在emitPartitionBatchNew方法中创建的。该方法不会更新元数据，于是保证了事务是处理相同的数据的。

17.4.2 分区 Spout 的执行器

PartitionedTridentSpoutExecutor类实现了ITridentSpout接口，它对IPartitioned TridentSpout进行了适配，使其能够被Trident执行。该类的代码如下：

```
1 public class PartitionedTridentSpoutExecutor implements ITridentSpout<Integer> {
2 IPartitionedTridentSpout _spout;
3
4 public PartitionedTridentSpoutExecutor(IPartitionedTridentSpout spout) {
5     _spout = spout;
6 }
7
8 public IPartitionedTridentSpout getPartitionedSpout() {
9     return _spout;
10 }
11
12 class Coordinator implements ITridentSpout.BatchCoordinator<Object> {
13     private IPartitionedTridentSpout.Coordinator _coordinator;
```

```
14
15     public Coordinator(Map conf, TopologyContext context) {
16         _coordinator = _spout.getCoordinator(conf, context);
17     }
18
19     @Override
20     public Object initializeTransaction(long txid, Object prevMetadata, Object currMetadata) {
21         if(currMetadata!=null) {
22             return currMetadata;
23         } else {
24             return _coordinator.getPartitionsForBatch();
25         }
26     }
27
28     @Override
29     public boolean isReady(long txid) {
30         return _coordinator.isReady(txid);
31     }
32     }
33
34    static class EmitterPartitionState {
35        public RotatingTransactionalState rotatingState;
36        public ISpoutPartition partition;
37
38        public EmitterPartitionState(RotatingTransactionalState s, ISpoutPartition p) {
39            rotatingState = s;
40            partition = p;
41        }
42    }
43
44    class Emitter implements ITridentSpout.Emitter<Object> {
45        private IPartitionedTridentSpout.Emitter _emitter;
46        private TransactionalState _state;
47        private Map<String, EmitterPartitionState> _partitionStates = new HashMap<String,
              EmitterPartitionState>();
48        private int _index;
49        private int _numTasks;
50
51        public Emitter(String txStateId, Map conf, TopologyContext context) {
52            _emitter = _spout.getEmitter(conf, context);
53            _state = TransactionalState.newUserState(conf, txStateId);
54            _index = context.getThisTaskIndex();
55            _numTasks = context.getComponentTasks(context.getThisComponentId()).size();
56        }
57
58        Object _savedCoordinatorMeta = null;
59
60
61        @Override
62        public void emitBatch(final TransactionAttempt tx, final Object coordinatorMeta,
63                final TridentCollector collector) {
64            if(_savedCoordinatorMeta == null || !_savedCoordinatorMeta.equals(coordinatorMeta)) {
65               List<ISpoutPartition> partitions = _emitter.getOrderedPartitions(coordinatorMeta);
66                _partitionStates.clear();
```

```
67              List<ISpoutPartition> myPartitions = new ArrayList();
68              for(int i=_index; i < partitions.size(); i+=_numTasks) {
69                  ISpoutPartition p = partitions.get(i);
70                  String id = p.getId();
71                  myPartitions.add(p);
72                  _partitionStates.put(id, new EmitterPartitionState(new Rotating
                        TransactionalState(_state, id), p));
73              }
74              _emitter.refreshPartitions(myPartitions);
75              _savedCoordinatorMeta = coordinatorMeta;
76          }
77          for(EmitterPartitionState s: _partitionStates.values()) {
78              RotatingTransactionalState state = s.rotatingState;
79              final ISpoutPartition partition = s.partition;
80              Object meta = state.getStateOrCreate(tx.getTransactionId(),
81                      new RotatingTransactionalState.StateInitializer() {
82                  @Override
83                  public Object init(long txid, Object lastState) {
84                      return _emitter.emitPartitionBatchNew(tx, collector, partition,
                            lastState);
85                  }
86              });
87              // it's null if one of:
88              // a) a later transaction batch was emitted before this, so we should skip
                      this batch
89              // b) if didn't exist and was created (in which case the StateInitializer
                      was invoked and
90              // it was emitted
91              if(meta!=null) {
92                  _emitter.emitPartitionBatch(tx, collector, partition, meta);
93              }
94          }
95      }
96
97      @Override
98      public void success(TransactionAttempt tx) {
99          for(EmitterPartitionState state: _partitionStates.values()) {
100             state.rotatingState.cleanupBefore(tx.getTransactionId());
101         }
102     }
103
104     @Override
105     public void close() {
106         _state.close();
107         _emitter.close();
108     }
109 }
110 }
```

- 第1行初始化其元数据类型为Integer。不过后面第12行和第44行又将接口中的元数据类型均实现为Object类型，故第1行处的元数据类型应为Object类型。
- 第12~32行实现了BatchCoordinator接口。

  - 第20~26行实现了initializeTransaction方法。注意如果参数currMetadata不为空，方法则直接返回。由于只有当事务进行重传时currMetadata才不为空，因此这保证了事务重传时具有相同的元数据。如果currMetadata为空，则通过getPartitionsForBatch方法获得分区的元数据，并且这部分数据应该是稳定的。
  - 第29~31行的isReady方法通过调用代理的isReady方法来实现。
- 第62~95行实现Emitter的emitBatch方法，这是该类的核心。第64行对输入的分区元数据进行判断，若为空或者发生变化，则更新ZooKeeper中存储的信息。注意，用户需要自定义equals方法来对元数据coordinatorMeta进行合理的比较。
- 第65行调用getOrderedPartitions方法，根据当前的分区元数据获得分区信息，然后根据节点的TaskId以及并行度，将分区均匀地分配到每个节点上。_partitionStates变量用来保存当前节点所负责的分区。注意，第72行为每个分区在ZooKeeper上分配了合理的位置，以存储每个分区的元数据。目录结构为：/spout-id/user/partition-id/txid。这也表明，事务在每个分区上都会有一个事务号与其相对应。
- 第74行调用refreshPartitions方法，用户可以根据分区的情况来更新自身的数据。例如，对Kafka Queue Brokers的连接，等等。
- 第77~94行依次对该节点所负责的每一个分区进行处理。第80~86行较为复杂，若事务是重传的，则元数据meta为事务第1次初始化时产生的的值，进一步如果getStateOrCreate返回值不为空，则直接调用emitPartitionBatch方法进行重传。若事务不是重传的，则利用第83行定义的init函数对事务进行初始化，emitPartitionBatchNew需要产生元数据。这里可以复用emitPartitionBatch的实现，返回值会被作为元数据进行存储。
- 第87~90行的注释很有趣。在模糊事务类型的 Spout中，事务的初始化并不要求强序，这使得后面的事务可以先于当前事务完成，即当前事务的数据已经被后一个事务完成了。此时，Trident需要将当前事务忽略掉。在IPartitionedTridentSpout中不会发生这种情况。
- 对于分区发生变更并且分区数目减少的情况，目前的系统实现在有些时候会导致上一个事务的数据不能被正确清理。例如处理顺序为P1, P2, S1, S2，P1代表事务1的处理阶段，而S1代表事务1的提交阶段，依此类推。若处理P2时数据的分区数目减少，_partitionStates已经被更新为新的分区信息，等到再去调用S1时，删除的分区已经不在_partitionStates中了，此时第98~102行代码将无法对元数据进行正确清理。

  可以看出，该类是在Emitter第一次遇到某个事务时创建的相应元数据，由于各个Emitter负责的分区没有交集，因此对于操作ZooKeeper是安全的。

17.5　模糊事务类型的 Spout 节点

类似于基本模糊事务类型的Spout节点，Trident中同样含有支持模糊事务类型的机制。

17.5.1 模糊事务类型的 Spout 接口

IOpaquePartitionedTridentSpout接口用于模糊事务类型Spout，即每个事务对应的元数据在同一事务的不同尝试之间是可以不同的。它的接口与IPartitionedTridentSpout基本相同，元数据的意义也是相同的。

```
public interface IOpaquePartitionedTridentSpout<Partitions, Partition extends ISpoutPartition,
     M> extends Serializable {
    public interface Coordinator<Partitions> {
        boolean isReady(long txid);
        Partitions getPartitionsForBatch();
        void close();
    }

    public interface Emitter<Partitions, Partition extends ISpoutPartition, M> {
        /**
         * Emit a batch of tuples for a partition/transaction.
         *
         * Return the metadata describing this batch that will be used as lastPartitionMeta
         * for defining the parameters of the next batch.
         */
        M emitPartitionBatch(TransactionAttempt tx, TridentCollector collector, Partition
             partition, M lastPartitionMeta);

        /**
         * This method is called when this task is responsible for a new set of partitions.
              Should be used
         * to manage things like connections to brokers.
         */
        void refreshPartitions(List<Partition> partitionResponsibilities);
        List<Partition> getOrderedPartitions(Partitions allPartitionInfo);
        void close();
    }

    Emitter<Partitions, Partition, M> getEmitter(Map conf, TopologyContext context);
    Coordinator getCoordinator(Map conf, TopologyContext context);
    Map getComponentConfiguration();
    Fields getOutputFields();
}
```

Emitter接口中只含有emitPartitionBatch方法，其输入参数为上一个事务所对应的元数据，并且没有当前事务的元数据。

17.5.2 模糊事务类型 Spout 的执行器

OpaquePartitionedTridentSpoutExecutor类实现了ICommitterTridentSpout接口，该接口向Emitter接口中新增加了commit方法。实现了ICommitterTridentSpout接口的节点会接收Master Coordinator的$commit流，并且会根据从该流收到的消息来更新元数据。理解何时产生、何时更新元数据对理解这个Spout是非常重要的。

本节主要分析一些关键方法的实现，首先来看Coordinator的代码：

```
public class Coordinator implements ITridentSpout.BatchCoordinator<Object> {
    @Override
    public Object initializeTransaction(long txid, Object prevMetadata, Object currMetadata) {
        return _coordinator.getPartitionsForBatch();
    }
```

在BatchCoordinator的initializeTransaction方法实现中，直接调用代理的_coordinator的getPartitionsForBatch方法来获得分区的元数据，而与当前的元数据currMetadata无关，这使得同一个事务的不同尝试可以对应于不同的分区。

emitBatch方法是实现该接口的核心实现方法，其部分代码与上一节中的PartitionedTridentSpoutExecutor是类似的，本节主要讨论它们的区别。相关代码如下：

```
1 public class Emitter implements ICommitterTridentSpout.Emitter {
2     @Override
3     public void emitBatch(TransactionAttempt tx, Object coordinatorMeta, TridentCollector
          collector) {
4         if(_savedCoordinatorMeta==null || !_savedCoordinatorMeta.equals(coordinatorMeta)) {
5             List<ISpoutPartition> partitions = _emitter.getOrderedPartitions(coordinatorMeta);
6              _partitionStates.clear();
7              List<ISpoutPartition> myPartitions = new ArrayList();
8              for(int i=_index; i < partitions.size(); i+=_numTasks) {
9                  ISpoutPartition p = partitions.get(i);
10                 String id = p.getId();
11                 myPartitions.add(p);
12                 _partitionStates.put(id, new EmitterPartitionState(new Rotating
                       TransactionalState(_state, id), p));
13             }
14             _emitter.refreshPartitions(myPartitions);
15             _savedCoordinatorMeta = coordinatorMeta;
16             _changedMeta = true;
17         }
18         Map<String, Object> metas = new HashMap<String, Object>();
19         _cachedMetas.put(tx.getTransactionId(), metas);
20 
21         Entry<Long, Map<String,Object>>entry=_cachedMetas.lowerEntry(tx.getTransactionId());
22         Map<String, Object> prevCached;
23         if(entry!=null) {
24             prevCached = entry.getValue();
25         } else {
26                prevCached = new HashMap<String, Object>();
27         }
28 
29         for(String id: _partitionStates.keySet()) {
30             EmitterPartitionState s = _partitionStates.get(id);
31             s.rotatingState.removeState(tx.getTransactionId());
32             Object lastMeta = prevCached.get(id);
33             if(lastMeta==null) lastMeta = s.rotatingState.getLastState();
34             Object meta = _emitter.emitPartitionBatch(tx, collector, s.partition, lastMeta);
35             metas.put(id, meta);
```

```
36         }
37     }
38 }
```

- 模糊事务类型的Spout中，事务元数据并不是在调用_emitter的emitPartitionBatch方法时就更新的，而是需要等到该事务已经被提交之后才能更新，这与基础的模糊事务类型的设计相一致。这里增加了_cachedMetas，用来存储当前遇到的事务的元数据。
- 第18~19行清空当前事务所对应的元数据。
- 第21~27行获得前一个事务所对应的元数据，以初始化lastMeta变量。
- 第31行对ZooKeeper中的数据进行清理。如果事务重传，其之前所对应的元数据都是无效的。例如，在提交的同时，若该事务已经超时，则元数据就会被更新，但事务还是要重新来做的。

接下来分析元数据是何时被更新的。commit方法的实现如下：

```
1 public void commit(TransactionAttempt attempt) {
2     // this code here handles a case where a previous commit failed, and the partitions
3     // changed since the last commit. This clears out any state for the removed partitions
4     // for this txid.
5     // we make sure only a single task ever does this. we're also guaranteed that
6     // it's impossible for there to be another writer to the directory for that partition
7     // because only a single commit can be happening at once. this is because in order for
8     // another attempt of the batch to commit, the batch phase must have succeeded in between.
9     // hence, all tasks for the prior commit must have finished committing (whether
           successfully or not)
10     if(_changedMeta && _index==0) {
11         Set<String> validIds = new HashSet<String>();
12         for(ISpoutPartition p: (List<ISpoutPartition>)_emitter.getOrderedPartitions
               (_savedCoordinatorMeta)) {
13             validIds.add(p.getId());
14         }
15         for(String existingPartition: _state.list("")) {
16             if(!validIds.contains(existingPartition)) {
17                 RotatingTransactionalState s = new RotatingTransactionalState(_state, existing
                      Partition);
18                 s.removeState(attempt.getTransactionId());
19             }
20         }
21         _changedMeta = false;
22     }
23
24     Long txid = attempt.getTransactionId();
25     Map<String, Object> metas = _cachedMetas.remove(txid);
26     for(String partitionId: metas.keySet()) {
27         Object meta = metas.get(partitionId);
28         _partitionStates.get(partitionId).rotatingState.overrideState(txid, meta);
29     }
30 }
```

- 当收到从$commit流发来的消息时，表明事务的处理阶段已经结束，事务提交Bolt已经要

开始调用finishBacth方法了。发来此时Trident认为已经到了一个恰当的时机，可以去更新元数据到ZooKeeper中，下一个事务的处理可以基于这个元数据来进行。

- 当然，在事务提交Bolt调用finishBatch方法时，仍然可以将事务标记为失败，这种情况下事务会进行重传。由于事务对应的元数据是可以变化的，所以目前的设计也是合理的。
- 第10~22行用来清理ZooKeeper中的数据。若前一个事务在处理阶段成功，但在提交时失败，同时下一个事务的处理阶段已经开始处理，则此时分区的元数据已经发生了变化。例如，某些分区可能在这个过程中被移除了，即最新的_savedCoordinatorMeta不再包含某些分区。然而在当前的事务中，已经在事务处理阶段重新将元数据放入了删除的分区。第17~18行对ZooKeeper进行了清理。第10行的条件意为若分区的元数据发生变化，同时保证只能是Task编号为第一的节点对元数据进行处理的条件下，才进行后面操作。多个节点同时删除ZooKeeper中的相同节点会导致异常。同时，Trident可以保证当前节点只有一个提交发生。

17.6 构建 Spout 节点

Spout节点在Trident中对应于一个新的流。本节分析Trident如何创建新的流。

17.6.1 TridentTopology 的 newStream 调用

利用传入的ITridentSpout类型的Spout节点，调用newStream可以创建一个新的流。在Trident中Spout类型为ITridentSpout，其他类型的Spout会被适配成为ITridentSpout。newStream的代码如下：

```
1 public Stream newStream(String txId, IRichSpout spout) {
2     return newStream(txId, new RichSpoutBatchExecutor(spout));
3 }
4
5 public Stream newStream(String txId, IBatchSpout spout) {
6      Node n = new SpoutNode(getUniqueStreamId(), spout.getOutputFields(), txId, spout,
          SpoutNode.SpoutType.BATCH);
7     return addNode(n);
8 }
9
10 public Stream newStream(String txId, ITridentSpout spout) {
11       Node n = new SpoutNode(getUniqueStreamId(), spout.getOutputFields(), txId, spout,
            SpoutNode.SpoutType.BATCH);
12      return addNode(n);
13   }
14
15   public Stream newStream(String txId, IPartitionedTridentSpout spout) {
16       return newStream(txId, new PartitionedTridentSpoutExecutor(spout));
17   }
18
19   public Stream newStream(String txId, IOpaquePartitionedTridentSpout spout) {
20       return newStream(txId, new OpaquePartitionedTridentSpoutExecutor(spout));
21   }
```

第10~13行对应的是真正创建节点代码，它创建了一个Spout节点。addNode函数在Trident的Topology中创建一个新的节点。newStream函数返回一个流对象。流对象是Trident中的重要概念，是构建Topology的基础，后面我们将对其进行详细讨论。

表17-1给出了Spout节点的适配关系。

表17-1 Spout节点类的适配关系

原始Spout类型	适配器或执行器
IRichSpout	RichSpoutBatchExecutor(spout)
IBatchSpout	BatchSpoutExecutor(spout) (TridentTopologyBuilder)
IPartitionedTridentSpout	PartitionedTridentSpoutExecutor(spout)
IOpaquePartitionedTridentSpout	OpaquePartitionedTridentSpoutExecutor(spout)

17.6.2 TridentTopology 中 newDRPCStream 调用

newDRPCStream会根据输入的DRPCSpout节点来获得一个流。DRPC技术主要用于查询，详情请参见第25章。DRPC流中的Spout节点较为特殊，其中一个事务中只含有一条消息。newDRPCStream的代码如下：

```
1 public Stream newDRPCStream(String function) {
2     return newDRPCStream(new DRPCSpout(function));
3  }
4
5  public Stream newDRPCStream(String function, ILocalDRPC server) {
6      DRPCSpout spout;
7      if(server==null) {
8          spout = new DRPCSpout(function);
9      } else {
10         spout = new DRPCSpout(function, server);
11     }
12     return newDRPCStream(spout);
13 }
14
15 private Stream newDRPCStream(DRPCSpout spout) {
16    // TODO: consider adding a shuffle grouping after the spout to avoid so much
          routing of the args/return-info all over the place
17     // (at least until its possible to just pack bolt logic into the spout itself)
18
19     Node n = new SpoutNode(getUniqueStreamId(), TridentUtils.getSingleOutput
          StreamFields(spout), null, spout, SpoutNode.SpoutType.DRPC);
20    Stream nextStream = addNode(n);
21    // later on, this will be joined back with return-info and all the results
22    return nextStream.project(new Fields("args"));
23 }
```

- 第15~23行代码用来创建一个DRPC流，DRPC流中包含了一个DRPC的Spout节点。
- 第22行，在得到DRPC流之后，添加一个映射节点，该节点的输出只有args列。

第 18 章 Trident的存储

在Storm中，Spout节点类型决定了Topology的类型，基本事务Spout节点以及模糊事务Spout节点可以实现消息的可靠传输，但并不能保证消息仅被处理一次，要想进一步保证这一点，就需要存储的支持。与Spout节点类型类似，存储也可以分为三种类型。图18-1给出了Spout节点类型与存储类型的对应关系，以及它们能否保证消息的“不多不少处理”这一语义（该图源自Storm官方网站）。

		State		
		Non-transactional	Transactional	Opaque transactional
Spout	Non-transactional	No	No	No
	Transactional	No	Yes	Yes
	Opaque transactional	No	No	Yes

图18-1　Spout节点与存储的对应关系

可以看出，即便使用事务类型Spout节点而不使用事务类型的存储，同样是不可能实现“不多不少的数据处理”（Exactly Once）这一语义的，必须要Spout节点与存储配合使用才能够。

注意，Storm中的存储并不是Trident特有的概念，但由于其实现主要在Trident的命名空间中发生，故本书将其与Trident一起讨论。

18.1　存储的基本接口

`State`接口是存储的基本接口。而`MapState`接口对应于数据为哈希表的情况，对于事务Topology，通常将事务序号作为键，事务的处理结果作为值，`MapState`接口更利于进行重操作。基本的类关系如图18-2所示。

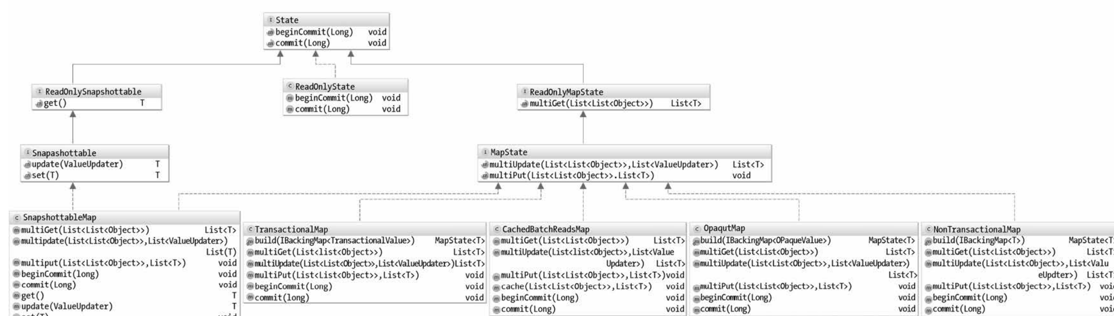


图18-2　State接口的类关系

下面讨论State接口以及MapState接口，相关代码如下：

```
public interface State {
    void beginCommit(Long txid); // can be null for things like partitionPersist occuring off a DRPC
       stream
    void commit(Long txid);
}

public interface ReadOnlyMapState<T> extends State {
    // certain states might only accept one-tuple keys - those should just throw an error
    List<T> multiGet(List<List<Object>> keys);
}

public interface MapState<T> extends ReadOnlyMapState<T> {
    List<T> multiUpdate(List<List<Object>> keys, List<ValueUpdater> updaters);
    void multiPut(List<List<Object>> keys, List<T> vals);
}
```

- State接口中定义了两个方法：beginCommit用来表示开始更新一个事务所对应的数据，commit方法则表示完成了对一个事务的更新。
- 并不是每一种存储的实现都需要用到这两个方法，例如在非事务类型的Topology中，数据将直接写入，并不需要调用beginCommit以及commit这两个方法。不过，这两个方法提供了事务开始和结束的调用时间点。
- 基本State接口中并未定义如何去更新数据，于是在ReadOnlyMapState接口中定义了一个multiGet方法。它的输入为List<Object>作为键的一组查询，输出为这些键对应的值。
- 在MapState接口中，额外的定义了multiUpdate和multiPut方法用于批量写入。目前，Storm中主要实现了MapState接口。

18.2　MapState 接口的实现

MapState接口是Storm中的重要接口，本节将分析该接口中相对重要实现。

18.2.1　非事务类型的存储

NonTransactionalMap实现了MapState接口，它也是MapState的最简单实现，用于非事务的情况。其代码如下：

```
1 public class NonTransactionalMap<T> implements MapState<T> {
2     public static <T> MapState<T> build(IBackingMap<T> backing) {
3        return new NonTransactionalMap<T>(backing);
4    }
5
6    IBackingMap<T> _backing;
7
8    protected NonTransactionalMap(IBackingMap<T> backing) {
9        _backing = backing;
10   }
11
12   @Override
13   public List<T> multiGet(List<List<Object>> keys) {
14       return _backing.multiGet(keys);
15   }
16
17   @Override
18   public List<T> multiUpdate(List<List<Object>> keys, List<ValueUpdater> updaters) {
19       List<T> curr = _backing.multiGet(keys);
20       List<T> ret = new ArrayList<T>(curr.size());
21       for(int i=0; i<curr.size(); i++) {
22           T currVal = curr.get(i);
23           ValueUpdater<T> updater = updaters.get(i);
24           ret.add(updater.update(currVal));
25       }
26       _backing.multiPut(keys, ret);
27       return ret;
28   }
29
30   @Override
31   public void multiPut(List<List<Object>> keys, List<T> vals) {
32       _backing.multiPut(keys, vals);
33   }
34
35   @Override
36   public void beginCommit(Long txid) {
37   }
38
39   @Override
40   public void commit(Long txid) {
41   }
42 }
```

- 第6行，IBackingMap<T>类型的_backing变量用于实际存储数据。
- 第2~4行的静态build方法将传入一个IBackingMap类型的对象，并构建一个NonTransactionalMap对象。用户可以自己实现新的IBackingMap对象，例如，若用户实现了一个基于文件或者数据库的IBackingMap对象，那么数据将会被存储到文件或者数据库中。

- NonTransactionalMap中beginCommit和commit的实现为空。
- 第18~28行实现multiUpdate方法，首先看一下输入参数的意义。
  - keys: List<Object>作为键，输入为一组键值。
  - updaters：每一个键都对应于这里的一个ValueUpdater对象。updaters的数目与keys的数目相同。ValueUpdater接口用来表示如何对已经存在值进行更新，通常它会含有一个更新算法。ValueUpdater接口是Trident对数据更新的抽象，本章其他小节将对其进行详细讨论。
- 第19行调用存储对象_backing的multiGet方法获得与键相对应的当前值。
- 第20~25行获得相应的ValueUpdater以及当前值，然后通过调用ValueUpdater的update方法计算得到更新后的值。
- 第26~27行将数据重新写回到_backing中，并返回更新后的值。
- 目前的实现中并未对输入和输出进行检查，例如若输入为5个键，在第19行去获取当前值时并不一定能保证返回5个值，这将导致不正确的状态。所以在IBackingMap的实现中需要注意返回值的数目。

18.2.2 事务类型的存储

TransactionalMap对应于事务Topology中的事务Spout，即同一个事务一定对应着同样的一组数据。该类的代码如下：

```
1 public class TransactionalMap<T> implements MapState<T> {
2     public static <T> MapState<T> build(IBackingMap<TransactionalValue> backing) {
3         return new CachedBatchReadsMap<T>(new TransactionalMap<T>(backing));
4     }
5
6     IBackingMap<TransactionalValue> _backing;
7     Long _currTx;
8
9     protected TransactionalMap(IBackingMap<TransactionalValue> backing) {
10        _backing = backing;
11    }
12
13    @Override
14    public List<T> multiGet(List<List<Object>> keys) {
15        List<TransactionalValue> vals = _backing.multiGet(keys);
16        List<T> ret = new ArrayList<T>(vals.size());
17        for(TransactionalValue v: vals) {
18            if(v!=null) {
19                ret.add((T) v.getVal());
20            } else {
21                ret.add(null);
22            }
23        }
24        return ret;
25    }
26
```

```
27     @Override
28     public List<T> multiUpdate(List<List<Object>> keys, List<ValueUpdater> updaters) {
29         List<TransactionalValue> curr = _backing.multiGet(keys);
30         List<TransactionalValue> newVals = new ArrayList<TransactionalValue>(curr.size());
31         List<T> ret = new ArrayList<T>();
32         for(int i=0; i<curr.size(); i++) {
33             TransactionalValue<T> val = curr.get(i);
34             ValueUpdater<T> updater = updaters.get(i);
35             TransactionalValue<T> newVal;
36             if(val==null) {
37               newVal = new TransactionalValue<T>(_currTx, updater.update(null));
38             } else {
39                 if(_currTx!=null && _currTx.equals(val.getTxid())) {
40                     newVal = val;
41                 } else {
42                     newVal = new TransactionalValue<T>(_currTx,updater.update(val.getVal()));
43                 }
44             }
45             ret.add(newVal.getVal());
46             newVals.add(newVal);
47         }
48         _backing.multiPut(keys, newVals);
49         return ret;
50     }
51
52     @Override
53     public void multiPut(List<List<Object>> keys, List<T> vals) {
54         List<TransactionalValue> newVals = new ArrayList<TransactionalValue>(vals.size());
55         for(T val: vals) {
56             newVals.add(new TransactionalValue<T>(_currTx, val));
57         }
58         _backing.multiPut(keys, newVals);
59     }
60
61     @Override
62     public void beginCommit(Long txid) {
63         _currTx = txid;
64     }
65
66     @Override
67     public void commit(Long txid) {
68         _currTx = null;
69     }
70 }
```

- TransactionalMap含有两个成员变量。
  - _backing。IBackingMap<TransactionalValue>类型。其中核心为TransactionalValue类，该类的定义如下：

```
public class TransactionalValue<T> {
    T val;
    Long txid;
    }
```

其中val为通用类型，为事务处理的结果，txid为事务序号，所以TransactionalValue表示为txid所对应的值。
 - _currTx表示当前的事务序号。
- 第28~50行实现multiUpdate方法，该方法是TransactionalMap的核心方法。
- 第36~38行对应于当前值为空的情况，此时将把ValueUpdater中含有的值作为返回值，并与当前的事务序号_currTx做比较。
- 第39~43行，若当前的_currTx与当前值中的事务序号相同，则直接将val赋值给newVal，表示并不对当前的值进行更新。这时每个事务所对应的数据是相同的。若_currTx与当前值中的txid不同，则调用ValueUpdater的update方法进行更新。由于只有在实现了ICommiter的Bolt中才能保证事务序号的强序关系，通常TransactionalMap只能用在事务提交Bolt中。
- 第61~64行实现beginCommit方法，将_currTx设置为输入值，该方法会在一个事务开始时被调用。
- 第67~69行实现commit方法，将_currTx置为空，该方法会在一个事务结束时被调用。

注意，multiUpdate需要将一个事务所对应的数据同时传入。其他的方法都是比较直观的，这里不再一一讨论。

18.2.3 模糊事务类型存储

OpaqueMap对应于事务Topology中的模糊事务Spout节点，表示同样的事务并不一定对应相同的数据，但是同一条消息一定只属于某一个事务。下面首先看一下OpaqueValue的定义及主要方法，它是OpaqueMap中存储的值，OpaqueValue的代码如下：

```
1 public class OpaqueValue<T> {
2     Long currTxid;
3     T prev;
4     T curr;
5
6     public OpaqueValue(Long currTxid, T val, T prev) {
7         this.curr = val;
8         this.currTxid = currTxid;
9         this.prev = prev;
10    }
11
12    public OpaqueValue<T> update(Long batchTxid, T newVal) {
13         T prev;
14         if(batchTxid!=null && batchTxid.equals(this.currTxid)) {
15             prev = this.prev;
16         } else {
17             prev = this.curr;
18         }
19         return new OpaqueValue<T>(batchTxid, newVal, prev);
20     }
21
22     public T get(Long batchTxid) {
23         if(batchTxid!=null && batchTxid.equals(currTxid)) {
```

```
24            return prev;
25        } else {
26            return curr;
27        }
28    }
29 }
```

- 与TransactionalValue相似，OpaqueValue含有txid、代表当前值的curr，额外的prev表示前一个事务所对应的值。
- 第12~20行定义update方法，这里curr都会被更新为newVal，主要看如何处理prev的值。若batchTxid与当前的currTxid相同，表示一个事务进行了重传，因此不更新prev的值；否则就将当前值curr赋值给prev。

OpaqueMap的方法实现与TransactionalMap的方法实现类似，这里主要分析一下其中的multi Update方法，相关代码如下：

```
1 public class OpaqueMap<T> implements MapState<T> {
2     @Override
3     public List<T> multiUpdate(List<List<Object>> keys, List<ValueUpdater> updaters) {
4         List<OpaqueValue> curr = _backing.multiGet(keys);
5         List<OpaqueValue> newVals = new ArrayList<OpaqueValue>(curr.size());
6         List<T> ret = new ArrayList<T>();
7         for(int i=0; i<curr.size(); i++) {
8             OpaqueValue<T> val = curr.get(i);
9             ValueUpdater<T> updater = updaters.get(i);
10            T prev;
11            if(val==null) {
12                prev = null;
13            } else {
14                prev = val.get(_currTx);
15            }
16            T newVal = updater.update(prev);
17            ret.add(newVal);
18            OpaqueValue<T> newOpaqueVal;
19            if(val==null) {
20                newOpaqueVal = new OpaqueValue<T>(_currTx, newVal);
21            } else {
22                newOpaqueVal = val.update(_currTx, newVal);
23            }
24            newVals.add(newOpaqueVal);
25        }
26        _backing.multiPut(keys, newVals);
27        return ret;
28    }
29 }
```

- 第10~15行获得当前的prev值。
- 第16~17行调用updater的update方法获得newVal，并将其放入到返回值中。
- 第19~24行获得newOpaqueVal，其中调用了OpaqueValue的update方法以完成更新。newVal与newOpaqueVal中curr值是相同的，区别反在于如何更新prev的值。

18.3 值的序列化方法

本节讨论与存储对应的3种值类型的序列化方法。Trident定义了Serializer接口，其代码如下：

```
public interface Serializer<T> extends Serializable {
    byte[] serialize(T obj);
    T deserialize(byte[] b);
}
```

用于对TransactionalValue、NonTransactionalValue和OpaqueValue进行序列化的类列举如下。

- JSONNonTransactionalSerializer：用于序列化通用类型T。
- JSONOpaqueSerializer：用于序列化OpaqueValue类。
- JSONTransactionalSerializer：用于序列化TransactionalValue类。

下面简单分析一下JSONOpaqueSerializer：

```
public class JSONOpaqueSerializer implements Serializer<OpaqueValue> {

    @Override
    public byte[] serialize(OpaqueValue obj) {
        List toSer = new ArrayList(3);
        toSer.add(obj.currTxid);
        toSer.add(obj.curr);
        toSer.add(obj.prev);
        try {
            return JSONValue.toJSONString(toSer).getBytes("UTF-8");
        } catch (UnsupportedEncodingException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public OpaqueValue deserialize(byte[] b) {
        try {
            String s = new String(b, "UTF-8");
            List deser = (List) JSONValue.parse(s);
            return new OpaqueValue((Long) deser.get(0), deser.get(1), deser.get(2));
        } catch (UnsupportedEncodingException e) {
            throw new RuntimeException(e);
        }
    }

}
```

- serialize的实现方法中，将currTxid、curr和prev放置于列表里，然后利用JSON将其序列化。
- deserialize的实现方法中，利用JSON反序列化得到列表，然后构建OpaqueValue对象。
- JSON序列化的结果是较为直观的，不过由于JSON序列化以及反序列化的效率较低，实际应用中通常采用更为高效的序列化方法。

18.4　数据更新接口

Trident 中采用 ValueUpdater 接口对存储的更新操作进行抽象：

```
public interface ValueUpdater<T> {
    T update(T stored);
}
```

其update方法的输入参数为通用类型T，stored表示更新前的值，返回值为更新后的值。ValueUpdater通常内含更新算法及输入。类关系如图18-3所示。

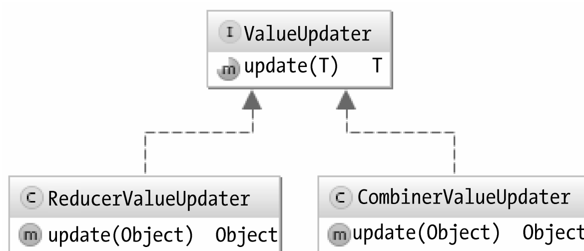


图18-3　数据更新接口的类关系

18.4.1　CombinerValueUpdater

CombinerValueUpdater实现了ValueUpdater接口，其代码如下：

```
public class CombinerValueUpdater implements ValueUpdater<Object> {
    Object arg;
    CombinerAggregator agg;

    public CombinerValueUpdater(CombinerAggregator agg, Object arg) {
        this.agg = agg;
        this.arg = arg;
    }

    @Override
    public Object update(Object stored) {
        if(stored==null) return arg;
        else return agg.combine(stored, arg);
    }
}
```

- CombinerValueUpdater内含一个CombinerAggregator类型的agg成员变量。update方法的输入参数stored表示当前值，类成员变量arg表示当前输入更新值。update方法会根据stored和arg产生一个新的值，若当前stored值为空，则直接返回arg值，否则调用agg的combine方法。

- CombinerValueUpdater中既包含了算法，又包含了将要用于更新的值。

18.4.2 ReducerValueUpdater

ReducerValueUpdater类与CombinerValueUpdater类似，它内含一个ReducerAggregator类型的成员变量agg，以及将要被用于更新的TridentTuple消息集合。该类的代码如下：

```
public class ReducerValueUpdater implements ValueUpdater<Object> {
    List<TridentTuple> tuples;
    ReducerAggregator agg;

    public ReducerValueUpdater(ReducerAggregator agg, List<TridentTuple> tuples) {
        this.agg = agg;
        this.tuples = tuples;
    }

    @Override
    public Object update(Object stored) {
        Object ret = stored;
        for(TridentTuple t: tuples) {
           ret =  this.agg.reduce(ret, t);
        }
        return ret;
    }
}
```

- 在update的实现方法中，依次调用了ReducerAggregator的reducer方法。
- 本书会在其他章节分析Aggregator和Combiner的接口及实现。

18.5 存储更新接口

StateUpdater接口用来对存储的更新操作进行抽象。而ValueUpdater则更为细粒度，它是对存储中每一个具体值的更新操作进行抽象。存储更新接口及其实现类的关系如图18-4所示。

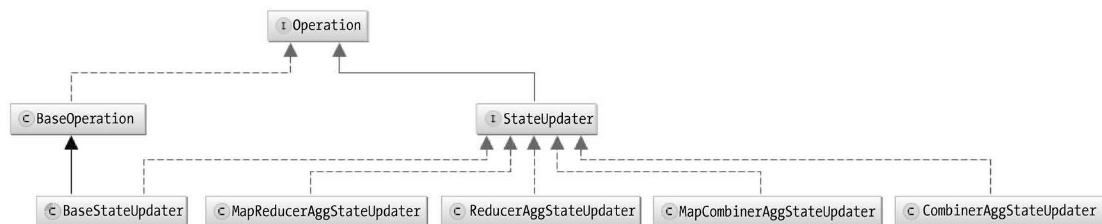


图18-4 存储更新接口的类关系

存储更新接口的定义分析如下，该接口继承自Operation接口。Operation接口将在其他章节进行讨论，它代表了Trident对操作的抽象。

```
public interface StateUpdater<S extends State> extends Operation {
    void updateState(S state, List<TridentTuple> tuples, TridentCollector collector);
}
```

该接口定义了updateState方法，其输入参数分析如下。

- state：实现State接口的对象，表示要进行更新的State对象。
- tuples：输入的TridentTuple消息列表。
- collector：TridentOutputCollector对象，用于发送数据。

相关的类实现中通常会使用相应的ValueUpdater对象，然后将其作为参数调用存储的multiUpdate方法。

18.5.1　ReducerAggStateUpdater

ReducerAggStateUpdater 实现了 StateUpdaer 接口，其代码如下：

```
public class ReducerAggStateUpdater implements StateUpdater<Snapshottable> {
    ReducerAggregator _agg;

    public ReducerAggStateUpdater(ReducerAggregator agg) {
        _agg = agg;
    }

    @Override
    public void updateState(Snapshottable state, List<TridentTuple> tuples,TridentCollector collector) {
        Object newVal = state.update(new ReducerValueUpdater(_agg, tuples));
        collector.emit(new Values(newVal));
    }
}
```

- ReducerAggStateUpdater中含有一个ReducerAggregator类型的_agg变量。在updateState的实现方法中创建了RecuderValueUpdater对象，然后调用存储的update方法。_agg将依次对输入的消息调用aggregate方法，并将最终的结果存储在存储对象中。
- ReducerAggStateUpdater最终会将得到的消息发送出去。
- CombinerAggStateUpdater与ReducerAggStateUpdater的实现很类似，只不过它的_agg为CombinerAggregator类型，其内部利用CombinerValueUpdater根据输入的消息对State进行更新。

18.5.2　MapReducerAggStateUpdater

MapReducerAggStateUpater主要用于分组聚集操作（Group By）之后对分组的结果进行更新，它进行聚集的键为分组的列。该类的代码如下：

```
1 public class MapReducerAggStateUpdater implements StateUpdater<MapState> {
2     ReducerAggregator _agg;
3     Fields _groupFields;
4     Fields _inputFields;
5     ProjectionFactory _groupFactory;
```

```
6     ProjectionFactory _inputFactory;
7     ComboList.Factory _factory;
8
9
10    public MapReducerAggStateUpdater(ReducerAggregator agg, Fields groupFields,Fields inputFields) {
11        _agg = agg;
12        _groupFields = groupFields;
13        _inputFields = inputFields;
14        _factory = new ComboList.Factory(groupFields.size(), 1);
15    }
16
17
18    @Override
19    public void updateState(MapState map, List<TridentTuple> tuples,TridentCollector collector) {
20        Map<List<Object>, List<TridentTuple>> grouped = new HashMap();
21
22        List<List<Object>> groups = new ArrayList<List<Object>>(tuples.size());
23        List<Object> values = new ArrayList<Object>(tuples.size());
24        for(TridentTuple t: tuples) {
25            List<Object> group = _groupFactory.create(t);
26            List<TridentTuple> groupTuples = grouped.get(group);
27            if(groupTuples==null) {
28                groupTuples = new ArrayList();
29                grouped.put(group, groupTuples);
30            }
31            groupTuples.add(_inputFactory.create(t));
32        }
33        List<List<Object>> uniqueGroups = new ArrayList(grouped.keySet());
34        List<ValueUpdater> updaters = new ArrayList(uniqueGroups.size());
35        for(List<Object> group: uniqueGroups) {
36            updaters.add(new ReducerValueUpdater(_agg, grouped.get(group)));
37        }
38        List<Object> results = map.multiUpdate(uniqueGroups, updaters);
39
40        for(int i=0; i<uniqueGroups.size(); i++) {
41            List<Object> group = uniqueGroups.get(i);
42            Object result = results.get(i);
43            collector.emit(_factory.create(new List[] {group, new Values(result) }));
44        }
45    }
46
47    @Override
48    public void prepare(Map conf, TridentOperationContext context) {
49        _groupFactory = context.makeProjectionFactory(_groupFields);
50        _inputFactory = context.makeProjectionFactory(_inputFields);
51    }
52 }
```

- _agg：ReducerAggregator类型对象，表示为聚集算法。
- _groupFields：进行分组的字段名。
- _groupFactory：ProjectionFactory类型，根据_groupFields来映射消息。
- _inputFileds：输入消息的字段名。

- _inputFactory：ProjectionFactory类型，根据_inputFields映射消息。
- _factory：ComboList.Factory对象。由两个部分构成，一部分为_groupFields，另外一部分为分组的结果，目前的结果是只能有1列。
- 第19~45行实现updateState方法。第22~23行定义的局部变量目前没有被用到。第20行定义局部变量grouped用来存储按照groupFields进行分组的消息，它的键为groupFields，值为具有相同groupFields的消息集合。第24~32行将输入的消息按照groupFields进行分组。
- 第33~37行根据分组的个数创建相应的ValueUpdater，每一个分组对应一个ValueUpdater。ReducerValueUpdater的输入为ReducerAggregator对象以及属于该分组的消息集合。该_agg将对此集合依次调用aggregate方法，结果被存储于存储对象中。第38行调用multiUpdate方法。第40~44行将结果消息输出。
- MapReducerAggStateUpater与ReducerAggStateUpdater的区别在于：前者是对MapState接口进行操作的，它的键为分组域；而后者更为通用，是对Snapshottable接口进行操作的，没有键的概念。
- MapCombinerAggStateUpdater与MapReducerAggStateUpater的实现类似，只不过它的_agg为CombinerAggregator类型，内部利用CombinerValueUpdater按照输入的消息对存储进行更新。

18.5.3 BaseStateUpdater

BaseStateUpdater同时实现了BaseOperation和StateUpdater两个接口，目前还仅仅处于想法阶段，并没有实际使用。目前StateUpdate的实现基本上是批处理模式的，即当一个事务处理结束之后才统一进行更新，那么可不可以逐步地对存储对象进行更新呢？这样设计才使得State接口中beginCommit方法和commit方法更具意义。这可能是Storm未来会改进的地方。

18.6 创建存储对象

StateFactory接口对如何创建一个State对象进行了抽象，相关代码如下：

```
public interface StateFactory extends Serializable {
    State makeState(Map conf, IMetricsContext metrics, int partitionIndex, int numPartitions);
}
```

这个接口含有一个makeState方法，用来创建一个新的State对象，其输入参数分析如下：

- conf：配置项。
- metrics：通常会传入上下文对象，例如TopologyContext对象。
- partitionIndex：当前State对应的索引。
- numPartitions：表示分区的数目。

例如，在SubtopologyBolt中初始化存储对象时传入的参数为：

```
State s = n.stateInfo.spec.stateFactory.makeState(conf, context, context.getThisTaskIndex(), this
    ComponentNumTasks);
```

```
conf: Topology在这个Component上面的配置。
metrics: 为TopologyContext对象context。
partitionIndex为当前Task的Id。
numPartitions为当前Task的并行度。
```

StateSpec类中包含了一个StateFactory对象。requiredNumPartitions将在TridentTopology类中进一步讨论，它用于计算SubTopologyBolt的并行度。该类的代码如下：

```
public class StateSpec implements Serializable {
    public StateFactory stateFactory;
    public Integer requiredNumPartitions = null;

    public StateSpec(StateFactory stateFactory) {
        this.stateFactory = stateFactory;
    }
}
```

用户需要实现接口IBackingMap、 State、ValueUpdater和StateFactory来完成自定义存储的实现，当然也可以只实现其中的一个部分来增加功能。

读者可以参考如下的类实现来学习如何实现相应的接口。

- State接口：storm.trident.testing.MemoryMapState
- IBackingMap接口：storm.trident.testing.MemoryMapState.MemoryMapStateBacking
- StateFactory接口：storm.trident.testing.MemoryMapState.Factory

第 19 章

Trident消息

19

在Trident中，一个Bolt节点中可能含有多个操作（Operation），各个操作之间需要进行消息传输。通常，操作或者产生新的域或者对原来的域进行过滤，若每次都对输入的消息进行复制，则效率不高。

Trident利用TridentTupleView对象对消息进行封装。例如，新产生的消息由两部分组成，一部分来自于输入，另外一部分则由计算得到。TridentTupleView并不会创建一个新的消息，而是将这两个部分合并，通过更新类内部的索引使得从外部看来如同一个消息一样。这样便节省了消息拷贝和新对象创建等方面的负担，因而提高了效率。

TridentTupleView是Trident中使用的消息类型，它继承自AbstractList<Object>，同时实现了接口TridentTuple。Trident采用了视图的方式进行了优化，不需要创建新的消息，而只是更新索引，并且索引在构建Bolt对象时就已经创建好了，这些都进一步地提升了效率。

本章将对Trident的消息及其构建方法进行讨论。

19.1 ValuePointer

ValuePointer类是用于描述TridentTupleView中索引的数据结构，每个ValuePointer对象对应于消息的一个域。它保存了索引用以访问数据，同时还提供了一些工具方法帮助完成索引的构建。该类的定义如下：

```
1 public class ValuePointer {
2     public static Map<String, ValuePointer> buildFieldIndex(ValuePointer[] pointers) {
3         Map<String, ValuePointer> ret = new HashMap<String, ValuePointer>();
4         for(ValuePointer ptr: pointers) {
5             ret.put(ptr.field, ptr);
6         }
7         return ret;
8     }
9
10    public static ValuePointer[] buildIndex(Fields fieldsOrder, Map<String,
          ValuePointer> pointers) {
11        if(fieldsOrder.size()!=pointers.size()) {
12          throw new IllegalArgumentException("Fields order must be same length as pointers map");
13        }
14         ValuePointer[] ret = new ValuePointer[pointers.size()];
```

```
15        List<String> flist = fieldsOrder.toList();
16        for(int i=0; i<fieldsOrder.size(); i++) {
17            ret[i] = pointers.get(fieldsOrder.get(i));
18        }
19        return ret;
20    }
21
22    public int delegateIndex;
23    protected int index;
24    protected String field;
25
26    public ValuePointer(int delegateIndex, int index, String field) {
27        this.delegateIndex = delegateIndex;
28        this.index = index;
29        this.field = field;
30    }
31 }
```

- 第22~24行定义了ValuePointer的成员变量。
  - delegateIndex为IPersistentVector类型的索引。TridentTupleView中可能含有多个IPersistentVector对象，该对象对应于数据存储。
  - index表示元素在某一个IPersistentVector集合中的索引。
  - field表示列的名称。
- 第10~20行，buildIndex函数将返回一个ValuePointer类型的对象数组。用户可以根据所在列的下标来访问元素。fieldsOrder表示输入列的顺序，下标可从0取到fieldsOrder.length-1。pointers表示列在输入消息中的真正索引。
- 第2~8行，buildFieldIndex函数通过ValuePointer数组返回一个从字段名指向ValulePointer的哈希表。

仔细阅读这个类，明白其中成员变量以及这两个工具方法的含义是很关键的。

19

19.2 Factory 接口及其实现

Factory接口用来描述如何创建Trident消息，本节将分析该接口及其主要实现类，相关代码如下：

```
public static interface Factory extends Serializable {
    Map<String, ValuePointer> getFieldIndex();
    List<String> getOutputFields();
    int numDelegates();
}
```

- numDelegates方法返回实际的IPersistentVector数目。
- getOutputFields方法则返回从外部看到的消息字段名，该字段名可能是由底层的几个IPersistentVector的字段名连接组合而成的。
- getFieldIndex方法将返回一个从字段名到ValulePointer的映射关系。ValulePointer用来访问底层数据。

19.2.1　ProjectionFactory

ProjectionFactory根据输入的parentFactory以及需要保留的字段名来重新构建一个消息。注意，ProjectionFactory并不产生新的消息，它只是根据输入的要保留的字段名来更新相应的索引。该工厂类主要用于映射操作，其代码如下：

```
1 public static class ProjectionFactory implements Factory {
2     Map<String, ValuePointer> _fieldIndex;
3     ValuePointer[] _index;
4     Factory _parent;
5
6     public ProjectionFactory(Factory parent, Fields projectFields) {
7         _parent = parent;
8         if(projectFields==null) projectFields = new Fields();
9         Map<String, ValuePointer> parentFieldIndex = parent.getFieldIndex();
10        _fieldIndex = new HashMap<String, ValuePointer>();
11        for(String f: projectFields) {
12            _fieldIndex.put(f, parentFieldIndex.get(f));
13        }
14        _index = ValuePointer.buildIndex(projectFields, _fieldIndex);
15    }
16
17    public TridentTuple create(TridentTuple parent) {
18        if(_index.length==0) return EMPTY_TUPLE;
19        else return new TridentTupleView(((TridentTupleView)parent)._delegates,
              _index, _fieldIndex);
20    }
21
22    @Override
23    public Map<String, ValuePointer> getFieldIndex() {
24        return _fieldIndex;
25    }
26
27    @Override
28    public int numDelegates() {
29        return _parent.numDelegates();
30    }
31
32    @Override
33    public List<String> getOutputFields() {
34        return indexToFieldsList(_index);
35    }
36 }
```

- 第6~15行，在构造函数中，初始化_fieldsIndex和_index成员变量。成员变量_fieldsIndex中包含从要保留的字段名到parent中的字段名的索引。然后，根据_fieldsIndex以及projectFields来构建_index索引。
- 第17~20行，根据parent的数据，以及通过要保留的列重新构建的_index和_fieldsIndex，返回一个新的TridentTupleView对象。

- 第28~30行，numDelegates返回_parent的numDelegates。这也表明了ProjectionFactory并不会产生新的列。

19.2.2　FreshOutputFactory

FreshOutputFactory根据输入的字段名和列值来产生一个新消息。该工厂方法对应于产生一条新消息的操作，其代码如下：

```
public static class FreshOutputFactory  implements Factory {
    Map<String, ValuePointer> _fieldIndex;
    ValuePointer[] _index;

    public FreshOutputFactory(Fields selfFields) {
        _fieldIndex = new HashMap<String, ValuePointer>();
        for(int i=0; i<selfFields.size(); i++) {
            String field = selfFields.get(i);
            _fieldIndex.put(field, new ValuePointer(0, i, field));
        }
        _index = ValuePointer.buildIndex(selfFields, _fieldIndex);
    }

    public TridentTuple create(List<Object> selfVals) {
        return new TridentTupleView(PersistentVector.EMPTY.cons(selfVals),_index, _fieldIndex);
    }

    @Override
    public Map<String, ValuePointer> getFieldIndex() {
        return _fieldIndex;
    }

    @Override
    public int numDelegates() {
        return 1;
    }

    @Override
    public List<String> getOutputFields() {
        return indexToFieldsList(_index);
    }
}
```

可以看出，该类的numDelegates方法返回值为1，其create方法需要传入构成TridentTupleView的值列表。_index和_fieldsIndex变量则由输入的字段名构建而来。

19.2.3　OperationOutputFactory

OperationOutputFactory将输入的消息与产生的消息连接以生成新的消息。顾名思义，这个类主要用于添加新域，其代码如下：

```
1 public static class OperationOutputFactory implements Factory {
2     Map<String, ValuePointer> _fieldIndex;
3     ValuePointer[] _index;
4     Factory _parent;
5
6     public OperationOutputFactory(Factory parent, Fields selfFields) {
7         _parent = parent;
8         _fieldIndex = new HashMap(parent.getFieldIndex());
9         int myIndex = parent.numDelegates();
10        for(int i=0; i<selfFields.size(); i++) {
11            String field = selfFields.get(i);
12            _fieldIndex.put(field, new ValuePointer(myIndex, i, field));
13        }
14        List<String> myOrder = new ArrayList<String>(parent.getOutputFields());
15
16        Set<String> parentFieldsSet = new HashSet<String>(myOrder);
17        for(String f: selfFields) {
18            if(parentFieldsSet.contains(f)) {
19                throw new IllegalArgumentException(
20                       "Additive operations cannot add fields with same name as already exists. "
21                        + "Tried adding " + selfFields + " to " + parent.getOutputFields());
22            }
23            myOrder.add(f);
24        }
25
26        _index = ValuePointer.buildIndex(new Fields(myOrder), _fieldIndex);
27    }
28
29    public TridentTuple create(TridentTupleView parent, List<Object> selfVals) {
30        IPersistentVector curr = parent._delegates;
31        curr = (IPersistentVector) RT.conj(curr, selfVals);
32        return new TridentTupleView(curr, _index, _fieldIndex);
33    }
34
35    @Override
36    public Map<String, ValuePointer> getFieldIndex() {
37        return _fieldIndex;
38    }
39
40    @Override
41    public int numDelegates() {
42        return _parent.numDelegates() + 1;
43    }
44
45    @Override
46    public List<String> getOutputFields() {
47        return indexToFieldsList(_index);
48    }
49 }
```

- 第6~27行的构造函数以parent为输入的工厂，selfFields表示将要产生的字段名，最终产生的消息将包含所有输入工厂中的列以及selfFilds中要包含的列。

- 第8行获得父工厂的fieldsIndex索引。
- 第9行，selfFields中的列将被放入一个新的delegate存储中，所以它将parent.numDelega tes作为自己的delegateIndex。注意，delegate的索引是从0开始的。然后在第41~43行中，numDelegates将在原来parent.numDelegates的基础上加1。
- 第10~13行将selfFields中的列加入到fieldsIndex中。
- 第16~24行构建用于输出的字段名列表，这里不允许selfFields中的字段名与parent中的字段名发生冲突。

19.2.4 RootFactory

RootFactory为操作的入口工厂，对输入的消息起到适配作用。它会根据输入消息产生一个TridentTupleView类型的消息，这个产生的消息可以被其他工厂方法使用。对象Tuple是在Storm中传输的对象。这里不再详细分析这个类的代码，仅列于此处以供参考：

```
public static class RootFactory implements Factory {
    ValuePointer[] index;
    Map<String, ValuePointer> fieldIndex;

    public RootFactory(Fields inputFields) {
        index = new ValuePointer[inputFields.size()];
        int i=0;
        for(String f: inputFields) {
            index[i] = new ValuePointer(0, i, f);
            i++;
        }
        fieldIndex = ValuePointer.buildFieldIndex(index);
    }

    public TridentTuple create(Tuple parent) {
        return new TridentTupleView(PersistentVector.EMPTY.cons(parent.getValues()), index,
            fieldIndex);
    }

    @Override
    public Map<String, ValuePointer> getFieldIndex() {
        return fieldIndex;
    }

    @Override
    public int numDelegates() {
        return 1;
    }

    @Override
    public List<String> getOutputFields() {
        return indexToFieldsList(this.index);
    }
}
```

19.3　消息工厂的例子

假定输入的消息含有列(A, B, C, D)，并依次进行一系列消息工厂方法的操作。具体的操作与结果如表19-1所示。其中(1, 0, A)用来表示一个ValuePointer对象，意义为存储1的第1列的字段名称为A。

表19-1　工厂操作示例

Factory	Num-Delegates	fieldsIndex	index	描　述
RootFactory	1	A->(0, 0, A)	0->(0, 0, A)	输入消息
		B->(0, 1, B)	1->(0, 1, B)	
		C->(0, 2, C)	2->(0, 2, C)	
		D->(0, 3, D)	3->(0, 3, D)	
ProjectionFactory	1	A->(0, 0, A)	0->(0, 0, A)	取A，C列
		C->(0, 2, C)	1->(0, 2, C)	
OperationOutputFactory	2	A->(0, 0, A)	0->(0, 0, A)	增加E， F列
		C->(0, 2, C)	1->(0, 2, C)	
		E->(1, 0, E)	2->(1, 0, E)	
		F->(1, 1, F)	3->(1, 1, F)	
ProjectionFactory	2	A->(0, 0, A)	0->(0, 0, A)	保留A，F列
		F->(1, 1, F)	1->(1, 1, F)	
FreshOutputFactory	1	G->(0, 0, G)	0->(0, 0, G)	产生G列

19.4　TridentTupleView

基于前面的讨论，我们接下来看类TridentTupleView。这个类实现了TridentTuple接口，该接口提供了很多可将输出的列值转化成目标类型的工具方法。同时，这个类还扩展了AbstractList类，这使得TridentTupleView消息可直接以Storm消息的形式被发送出去。该类的代码如下：

```
1 //extends abstractlist so that it can be emitted directly as Storm tuples
2 public class TridentTupleView extends AbstractList<Object> implements TridentTuple {
3     ValuePointer[] _index;
4     Map<String, ValuePointer> _fieldIndex;
5     IPersistentVector _delegates;
6
7     public static TridentTupleView EMPTY_TUPLE = new TridentTupleView(null, new
          ValuePointer[0], new HashMap());
8
9     // index and fieldIndex are precomputed, delegates built up over many operations
          using persistent data structures
10    public TridentTupleView(IPersistentVector delegates, ValuePointer[] index,
          Map<String, ValuePointer> fieldIndex) {
11        _delegates = delegates;
12        _index = index;
13        _fieldIndex = fieldIndex;
```

```
14    }
15
16    @Override
17    public Object get(int i) {
18        return getValue(i);
19    }
20
21    @Override
22    public Object getValue(int i) {
23        return getValueByPointer(_index[i]);
24    }
25
26    @Override
27    public Object getValueByField(String field) {
28        return getValueByPointer(_fieldIndex.get(field));
29    }
30
31    private Object getValueByPointer(ValuePointer ptr) {
32        return ((List<Object>)_delegates.nth(ptr.delegateIndex)).get(ptr.index);
33    }
34 }
```


- 第17~19行代码覆盖AbstractList的get方法，它将调用getValue方法，这是可以把TridentTupleView作为消息进行发送的基础。
- 第31~33行代码根据输入的ValulePointer对象获得数据。首先根据ptr.delegateIndex获得delegate引用，然后根据ptr.index获取前面delegate引用上的第ptr.index个元素。
- 第27~29行代码根据输入的字段名field以及_fieldIndex映射关系找到其所对应的ValuePointer，并调用getValueByPointer方法。
- 第22~24行代码根据输入列的下标i以及_index找到与第i列相对应的索引，然后调用getValueByPointer方法获得列值。这样，TridentTupleView便既支持以字段名的方式访问元素也支持通过下标的方式访问元素了。

19.5 ComboList

ComboList的设计与TridentTupleView非常相似，它将输入的列表进行连接并创建索引。TridentTupleView主要用于各个操作之间的消息传输，而ComboList仅为Trident中的工具方法。

```
1 public class ComboList extends AbstractList<Object> {
2     public static class Factory implements Serializable {
3         Pointer[] index;
4         int[] sizes;
5
6         public Factory(int... sizes) {
7             this.sizes = sizes;
8             int total = 0;
9             for(int size: sizes) {
10                total+=size;
11            }
```

```
12            index = new Pointer[total];
13            int i=0;
14            int j=0;
15            for(int size: sizes) {
16                for(int z=0; z<size; z++) {
17                    index[j] = new Pointer(i, z);
18                    j++;
19                }
20                i++;
21            }
22        }
23
24        public ComboList create(List[] delegates) {
25            if(delegates.length!=sizes.length) {
26               throw new RuntimeException("Expected " + sizes.length + " lists, but instead got "
                     + delegates.length + " lists");
27               }
28               for(int i=0; i<delegates.length; i++) {
29                   List l = delegates[i];
30                   if(l==null || l.size() != sizes[i]) {
31                       throw new RuntimeException("Got unexpected delegates to ComboList: " +
                             ToStringBuilder.reflectionToString(delegates));
32                   }
33               }
34               return new ComboList(delegates, index);
35           }
36       }
37
38       private static class Pointer implements Serializable {
39           int listIndex;
40           int subIndex;
41
42           public Pointer(int listIndex, int subIndex) {
43               this.listIndex = listIndex;
44               this.subIndex = subIndex;
45           }
46
47       }
48
49       Pointer[] _index;
50       List[] _delegates;
51
52       public ComboList(List[] delegates, Pointer[] index) {
53           _index = index;
54           _delegates = delegates;
55       }
56
57       @Override
58       public Object get(int i) {
59           Pointer ptr = _index[i];
60           return _delegates[ptr.listIndex].get(ptr.subIndex);
61       }
62
63       @Override
```

```
64        public int size() {
65            return _index.length;
66        }
67 }
```

- 第38~47行代码定义私有类Pointer对象。listIndex为第一层索引，表示所在的列表索引；subIndex为第二层索引，表示在某一个列表内部的索引。
- 第2~22行代码定义了内部类Factory。Index是Pointer的对象，用来访问列表中的元素；sizes成员变量用来表示每一个列表的列数。该类主要用于对输入进行有效性验证。
- 第6~22行代码，在Factory的构造函数中，利用输入的每个列表的列数目构建索引。
- 第58~61行代码实现了ComoList的get方法，它会根据输入列的下标找到Pointer对象，再根据Pointer对象访问具体的元素。
- 与TridentTupleView相比，ComboList更加灵活，它可以接收多个列表作为输入，使它们从外界看来就如同一个列表。同时，ComobList继承自AbstractList，所以也可以作为Storm的消息进行发送。

第 20 章 Trident操作与处理节点

20

Trident中定义了较多的操作接口，用户通常需要实现这些接口来具体地使用它们，详情可参见https://github.com/nathanmarz/storm/wiki/Trident-API-Overview。

流是Trident数据模型中的核心概念，一个流经过分区操作，会分布在集群中的不同节点上。这些针对同一个流的操作可以在集群中并行起来。根据具体方式的不同，目前有5种类型的流操作，如表20-1所示。

表20-1　流操作类型

类　　型	描　　述
局部操作（Local Operation）	应用于一个分区内部，不会产生网络传输，通常被放置于Bolt节点中执行
分区操作（Repartition Operation）	对一个流进行重新分区，但并不改变内容，会导致网络传输
聚集操作（Aggregation Operation）	对分区中数据进行聚集，会产生网络传输
分组操作（Grouped Operation）	在分组流（Grouped Stream）上进行的操作
合并连接操作（Merge & Join）	流之间合并以及连接

本章及接下来的两章将介绍Trident的操作接口以及对流的操作。本章主要介绍聚集器接口、Trident中的处理节点，以及如何在Trident中执行聚集器类型的操作。

20.1　操作的基本接口

`Operation`接口为Trident中的操作接口，而`Filter`、`Function`和`Aggregator`接口则是用户实际使用的接口。操作接口的类关系如图20-1所示，它们的基本含义见表20-2。

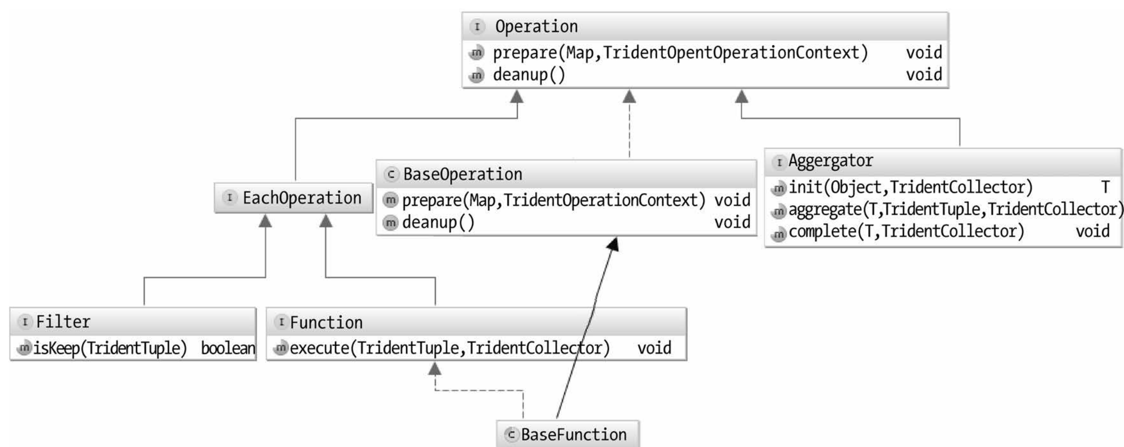


图20-1 操作接口的关系

表20-2 操作接口描述

接 口	主要方法	描 述
Operation	prepare cleanup	对Trident操作的抽象 TridentOperationContext含有该组件的输入信息以及Topology的上下文信息
EachOperation	无新方法	对每条数据均进行处理的抽象
Filter	isKeep	对于输入的消息，调用isKeep方法判断是否输出或丢掉
Function	execute	一个函数对输入的消息进行处理，可以有一条或者多条消息输出，新产生出来的列被放在原有列的后面，即Function不会使列数减少
BaseOperation	无新方法	
Aggregator	init aggregate complete	聚集操作的抽象。Batch开始时调用init方法，聚集过程中调用aggregate方法，聚集结束时调用complete方法

20.2 Aggregator 实现

Aggregator接口是关于聚集操作的核心接口，其实现是本章的重点，关于Aggregator的主要实现类关系如图20-2所示。

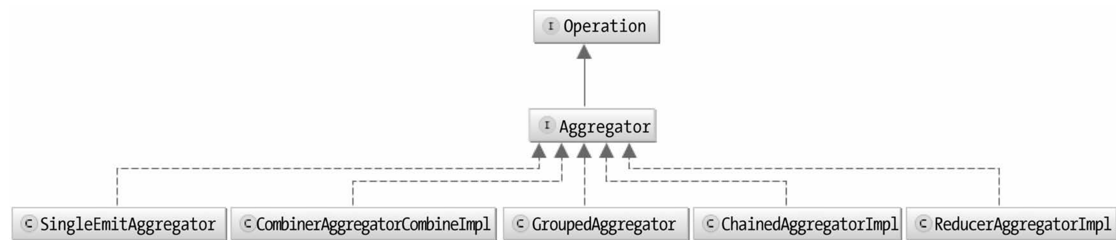


图20-2 聚集接口的实现类关系

本节将分别讨论这些聚集器的实现以及主要用途。

20.2.1 GroupedAggregator

流的分组操作（Group By）即对流中的数据按照某些域进行分组。在具体的某个分组上，可以使用GroupedAggregator类对该分组中的消息进行聚集操作。该类的定义如下所示：

```
1 public class GroupedAggregator implements Aggregator<Object[]> {
2     ProjectionFactory _groupFactory;
3     ProjectionFactory _inputFactory;
4     Aggregator _agg;
5     ComboList.Factory _fact;
6     Fields _inFields;
7     Fields _groupFields;
8 
9     public GroupedAggregator(Aggregator agg, Fields group, Fields input, int outSize) {
10        _groupFields = group;
11        _inFields = input;
12        _agg = agg;
13        int[] sizes = new int[2];
14        sizes[0] = _groupFields.size();
15        sizes[1] = outSize;
16        _fact = new ComboList.Factory(sizes);
17    }
18 
19    @Override
20    public void prepare(Map conf, TridentOperationContext context) {
21        _inputFactory = context.makeProjectionFactory(_inFields);
22        _groupFactory = context.makeProjectionFactory(_groupFields);
23        _agg.prepare(conf, new TridentOperationContext(context, _inputFactory));
24    }
25 
26    @Override
27    public Object[] init(Object batchId, TridentCollector collector) {
28        return new Object[] {new GroupCollector(collector, _fact), new HashMap(), batchId};
29    }
30 
31    @Override
32    public void aggregate(Object[] arr, TridentTuple tuple, TridentCollector collector) {
33        GroupCollector groupColl = (GroupCollector) arr[0];
34        Map<List, Object> val = (Map) arr[1];
35        TridentTuple group = _groupFactory.create((TridentTupleView) tuple);
36        TridentTuple input = _inputFactory.create((TridentTupleView) tuple);
37        Object curr;
38        if(!val.containsKey(group)) {
39            curr = _agg.init(arr[2], groupColl);
40            val.put((List) group, curr);
41        } else {
42            curr = val.get(group);
43        }
44        groupColl.currGroup = group;
45        _agg.aggregate(curr, input, groupColl);
```

```
46
47    }
48
49    @Override
50    public void complete(Object[] arr, TridentCollector collector) {
51        Map<List, Object> val = (Map) arr[1];
52        GroupCollector groupColl = (GroupCollector) arr[0];
53        for(Entry<List, Object> e: val.entrySet()) {
54            groupColl.currGroup = e.getKey();
55            _agg.complete(e.getValue(), groupColl);
56        }
57    }
58
59    @Override
60    public void cleanup() {
61        _agg.cleanup();
62    }
63 }
```

- 第1行，GroupedAggregator实现了Aggegator接口，聚集的结果被放在一个对象数组中。从第27~29行定义的init方法可以看出，该对象数组含有3个元素。
  - GroupCollector：实现TridentCollector接口。其消息由两部分组成，分别为用于分组的列和用于存储聚集结果的列。
  - 哈希表对象：用来存储聚集结果，键为分组的列值，值为聚集结果。
  - 事务序号：事务的序号标识符。
- 类的成员变量分析如下。
  - _groupedFactory：根据分组的列获得的相应的列值。
  - _inputFactory：根据需要的输入列获得的相应的列值。
  - _agg：内嵌的Aggregator，用于对属于同一个分组的消息进行聚集。
  - _fact：用于描述输出，它由两个部分构成，即分组的列和新产生的列。
  - _inputFields：输入的字段名。
  - _groupFields：用于分组的字段名。
- 第32~47行实现了核心的aggregate方法。第35~36行利用工厂方法获得输入消息和进行分组的消息（group为消息input的某些列）。第37~43行以分组消息为键获得当前分组上的聚集值，若目前还没有聚集值，则调用_agg的init方法进行初始化，表示第一次遇到某个分组中的元素。
- 第45行调用_agg的aggregate方法，传入的参数为当前消息和当前分组的聚集值。
- 第50~57行实现了complete方法，这表示属于某一个事务的处理结束，也意味着在这个分组上面的聚集已经完成。
- 第51行的val变量包含了以分组值作为键的结果。第55行依次在每个分组上调用_agg的complete方法。

20.2.2 ChainedAggregatorImpl

ChainedAggregatorImpl类中可以含有多个聚集器，它会依次调用这些聚集器，并在最后通过叉积（Cross Join）的方式将各个聚集器的聚集结果发送出去。

ChainedResult类用来存储聚集的结果，其定义如下：

```
public class ChainedResult {
    Object[] objs;
    TridentCollector[] collectors;

    public ChainedResult(TridentCollector collector, int size) {
        objs = new Object[size];
        collectors = new TridentCollector[size];
        for(int i=0; i<size; i++) {
            if(size==1) {
                collectors[i] = collector;
            } else {
                collectors[i] = new CaptureCollector();
            }
        }
    }
}
```

ChainedResult类包含两个成员变量，objs用来存储聚集过程的中间结果，collectors被聚集器用来发送结果。从它的构造函数可以看出，如果要使用多个聚集器时，collector将被初始化为CaptureCollector。

ChainedAggregatorImpl类包含了一组聚集器，对于输入的消息，该类将依次调用这些聚集器，并在最后将每一个聚集器的结果通过叉积的方式发送出去。下面我们来分析一下ChainedAggregatorImpl的实现，其代码如下：

```
1 public class ChainedAggregatorImpl implements Aggregator<ChainedResult> {
2     Aggregator[] _aggs;
3     ProjectionFactory[] _inputFactories;
4     ComboList.Factory _fact;
5     Fields[] _inputFields;
6
7
8    public ChainedAggregatorImpl(Aggregator[] aggs, Fields[] inputFields,ComboList.Factory fact) {
9        _aggs = aggs;
10       _inputFields = inputFields;
11       _fact = fact;
12       if(_aggs.length!=_inputFields.length) {
13           throw new IllegalArgumentException("Require input fields for each aggregator");
14       }
15   }
16
17   public void prepare(Map conf, TridentOperationContext context) {
18       _inputFactories = new ProjectionFactory[_inputFields.length];
19       for(int i=0; i<_inputFields.length; i++) {
```

```
20          _inputFactories[i] = context.makeProjectionFactory(_inputFields[i]);
21          _aggs[i].prepare(conf, new TridentOperationContext(context,_inputFactories[i]));
22      }
23  }
24
25  public ChainedResult init(Object batchId, TridentCollector collector) {
26      ChainedResult initted = new ChainedResult(collector, _aggs.length);
27      for(int i=0; i<_aggs.length; i++) {
28          initted.objs[i] = _aggs[i].init(batchId, initted.collectors[i]);
29      }
30      return initted;
31  }
32
33  public void aggregate(ChainedResult val, TridentTuple tuple, TridentCollector collector) {
34      val.setFollowThroughCollector(collector);
35      for(int i=0; i<_aggs.length; i++) {
36          TridentTuple projected = _inputFactories[i].create((TridentTupleView) tuple);
37          _aggs[i].aggregate(val.objs[i], projected, val.collectors[i]);
38      }
39  }
40
41  public void complete(ChainedResult val, TridentCollector collector) {
42      val.setFollowThroughCollector(collector);
43      for(int i=0; i<_aggs.length; i++) {
44          _aggs[i].complete(val.objs[i], val.collectors[i]);
45      }
46      if(_aggs.length > 1) { // otherwise, tuples were emitted directly
47          int[] indices = new int[val.collectors.length];
48          for(int i=0; i<indices.length; i++) {
49              indices[i] = 0;
50          }
51          boolean keepGoing = true;
52          //emit cross-join of all emitted tuples
53          while(keepGoing) {
54              List[] combined = new List[_aggs.length];
55              for(int i=0; i< _aggs.length; i++) {
56                  CaptureCollector capturer = (CaptureCollector) val.collectors[i];
57                  combined[i] = capturer.captured.get(indices[i]);
58              }
59              collector.emit(_fact.create(combined));
60              keepGoing = increment(val.collectors, indices, indices.length - 1);
61          }
62      }
63  }
64
65  //return false if can't increment anymore
66  private boolean increment(TridentCollector[] lengths, int[] indices, int j) {
67      if(j==-1) return false;
68      indices[j]++;
69      CaptureCollector capturer = (CaptureCollector) lengths[j];
70      if(indices[j] >= capturer.captured.size()) {
71          indices[j] = 0;
72          return increment(lengths, indices, j-1);
73      }
```

```
74        return true;
75    }
76
77    public void cleanup() {
78        for(Aggregator a: _aggs) {
79            a.cleanup();
80        }
81    }
```

- _aggs为ChainedAggregatorImpl中包含的聚集器链。
- _inputFields为每一个聚集器所对应的输入。
- _ inputFactories为负责根据inputFields来产生消息的工厂，其个数应与_inputFields和_aggs的个数相同。
- _fact用来存储最终结果，它是所有聚集器结果的合并。
- 当_aggs仅含有一个Aggregator时，ChainedAggregatorImpl将退化为普通的聚集器。
- 第8~15行定义了构造函数，其中需要为每一个聚集器指定一个输入的字段名。
- 第17~23行，在prepare方法中，根据聚集器输入的Fields定义一个工厂，该工厂可根据Fields将输入的原始消息转换成聚集器输入所需的消息类型，主要是对输入列进行过滤以及将字段名重新排列等。最后调用聚集器自身的prepare方法。
- 第25~31行是init方法的实现。这里将调用每一个聚集器的init方法，并将结果存储于ChainedResult中的objs对象中，它是聚集过程的中间结果。
- 第33~39行为aggregate方法的具体实现。这里将对输入的消息逐一调用聚集器的aggregate方法。
- 第41~63行代码实现complete方法。其复杂性在于，若存在多个聚集器时，将输出所有结果的叉积作为最终的结果。例如：

  ```
  Aggr1: A1, A2
  Aggr2: B1, B2
  Aggr3: C1
  ```

  其中A1、A2表示为Aggr1的两个输出结果。
  那么，最终的结果将含有4个消息，并且每个消息包含三列，依次为：

  ```
  A1, B1, C1
  A1, B2, C1
  A2, B1, C1
  A2, B2, C1
  ```

- 第47~48行代码定义了indices数组并初始化为0， 用来表示每个聚集器结果的下标。第54~59行代码根据indices中的数值取出元素并发送出去。
- 第66~75行代码调整了indices中的下标位置，其基本思想是，如果最后一个聚集器的下标已经超过了其元素个数，则开始调整前一个聚集器的下标。不过在代码的第60行，始终都是传入聚集器数目少1作为下标调整的初始值，这保证了最终结果是所有聚集器结果

的叉积（Cross Join）。该算法较为精巧，由于所包含的聚集器的数目并不确定，多重循环算法在这里是不合适的。

20.2.3 SingleEmitAggregator

SingleEmitAggregator类用于应对聚集集合为空的情况。此时只需要其中的一个节点向目标节点发送消息，且该消息含有聚合结果的初始值即可。BatchToPartition接口是在SingleEmitAggregator类中定义的，下面来看该接口的定义及其两个实现。

```
public static interface BatchToPartition extends Serializable {
    int partitionIndex(Object batchId, int numPartitions);
}
```

BatchToPartition接口中的方法partitionIndex，可根据batchId以及分区数目获得当前可以发送消息的分区ID。目前numPartitions为Task的索引，表示通过该节点可以向目标节点发送消息。Trident中有两个实现了该接口的类。

GlobalBatchToPartition类始终使用第一个节点来发送消息，其定义如下：

```
public class GlobalBatchToPartition implements SingleEmitAggregator.BatchToPartition {
    public int partitionIndex(Object batchId, int numPartitions) {
        // TODO: take away knowledge of storm's internals here
        return 0;
    }
}
```

而IndexHashBatchToPartition类则利用事务序号的哈希值来计算由哪个节点发送该消息，其定义如下：

```
public class IndexHashBatchToPartition implements
SingleEmitAggregator.BatchToPartition {
    public int partitionIndex(Object batchId, int numPartitions) {
        return IndexHashGrouping.objectToIndex(batchId, numPartitions);
    }
}
```

最后，来看SingleEmitAggregator的实现，其代码如下：

```
1    public class SingleEmitAggregator implements Aggregator<SingleEmitState> {
2        static class SingleEmitState {
3            boolean received = false;
4            Object state;
5            Object batchId;
6            public SingleEmitState(Object batchId) {
7                this.batchId = batchId;
8            }
9        }
10
11       Aggregator _agg;
```

20

```
12      BatchToPartition _batchToPartition;
13
14      public SingleEmitAggregator(Aggregator agg, BatchToPartition batchToPartition) {
15          _agg = agg;
16          _batchToPartition = batchToPartition;
17      }
18
19      @Override
20      public SingleEmitState init(Object batchId, TridentCollector collector) {
21           return new SingleEmitState(batchId);
22      }
23
24      @Override
25      public void aggregate(SingleEmitState val, TridentTuple tuple, TridentCollector collector) {
26          if(!val.received) {
27               val.state = _agg.init(val.batchId, collector);
28               val.received = true;
29          }
30          _agg.aggregate(val.state, tuple, collector);
31      }
32
33      @Override
34      public void complete(SingleEmitState val, TridentCollector collector) {
35          if(!val.received) {
36              if(this.myPartitionIndex == _batchToPartition.partitionIndex(val.batchId, this.
                     totalz Partitions)) {
37                val.state = _agg.init(val.batchId, collector);
38                _agg.complete(val.state, collector);
39              }
40          } else {
41              _agg.complete(val.state, collector);
42          }
43      }
44
45      int myPartitionIndex;
46      int totalPartitions;
47
48      @Override
49      public void prepare(Map conf, TridentOperationContext context) {
50          _agg.prepare(conf, context);
51          this.myPartitionIndex = context.getPartitionIndex();
52          this.totalPartitions = context.numPartitions();
53      }
54 }
```

- SingleEmitState为SingleEmitAggregator进行聚集的数据结构。其中，received表示聚集的集合是否含有数据，batchId表示事务序号，state为实际的聚集结果对象。
- myPartitionIndex为当前Task的索引，totalPartitions为Task的并行度。
- 第24~31行的aggregate方法被调用时，表明该集合是有数据的，这里会设置received为true。
- 第27行调用_agg的init方法，此处为第一次收到该集合中的数据。
- 第34~43行的complete方法实现中，若received为false，表明集合为空，则判断是否由当

前的节点来发送集合初始化得到消息。Complete方法会调用partitionIndex方法来获得应由哪个节点发送该消息。

- 第37~38行调用_agg的init方法以及complete方法。
- 在Trident中，有时候上游的聚集节点没有产生聚集结果，但下游节点仍然需要其发送消息（即便是初始化消息）来完成操作（例如，连接操作等）。

20.3　用户接口及其实现

聚集器接口是Trident的基本接口，但它们并不能满足所有的需求。因此，Trident定义了ReducerAggregator和CombinerAggregator两个新的接口，用以丰富聚集操作的类型。由于这两种接口都不是继承自Aggregator接口的，Trident采用了与前面介绍内容相类似的技术，通过分别实现它们的执行类来完成接口的适配。

20.3.1　ReducerAggregator接口及其实现

ReducerAggregator接口是对聚集操作的一种抽象，这个抽象更加切合实际应用。它的reduce函数需传入当前的聚集结果以及要进行聚集的消息。该接口的定义如下：

```
public interface ReducerAggregator<T> extends Serializable {
    T init();
    T reduce(T curr, TridentTuple tuple);
}
```

- init函数返回一个T类型的值。reduce方法接收T类型的当前值curr，以及一个TridentTuple类型的消息，返回值也为一个T类型的值。
- 对于属于同一个事务的消息，可通过不断调用reduce方法来达到聚集消息的目标。用户通常通过实现ReducerAggregator接口来实现用户逻辑。

ReducerAggregatorImpl类实现了Aggregator接口，它可作为ReducerAggregator的执行器，其定义如下：

```
1  public class ReducerAggregatorImpl implements Aggregator<Result> {
2      ReducerAggregator _agg;
3
4      public ReducerAggregatorImpl(ReducerAggregator agg) {
5          _agg = agg;
6      }
7
8      public void prepare(Map conf, TridentOperationContext context) {
9      }
10
11     public Result init(Object batchId, TridentCollector collector) {
12         Result ret = new Result();
13         ret.obj = _agg.init();
14         return ret;
15     }
```

```
16
17     public void aggregate(Result val, TridentTuple tuple, TridentCollector collector) {
18         val.obj = _agg.reduce(val.obj, tuple);
19     }
20
21     public void complete(Result val, TridentCollector collector) {
22         collector.emit(new Values(val.obj));
23     }
24 }
```

- Result类中只包含一个Object类型的成员变量obj，用来存储聚集结果。
- 第11~15行中，init的方法实现调用了_agg的init方法，并将返回值存储到Result类中的obj对象中。
- 第17~19行中，aggregate的实现调用了_agg的reduce方法。这里可以看出ReducerAggregator接口与Aggregator接口的区别，即reduce方法中并不需要传入TridentCollector，也就表明在聚集的过程中并不会向外发送消息。

20.3.2　CombinerAggregator接口及其实现

CombinerAggregator接口是对聚集的另外一种抽象。它含有一个combine方法，该方法的收入参数分别对应两个初步的聚集结果。接口的定义如下：

```
public interface CombinerAggregator<T> extends Serializable {
    T init(TridentTuple tuple);
    T combine(T val1, T val2);
    T zero();
}
```

- 该接口会对事务中的每一条消息调用init方法，得到初步的聚集结果T，然后在此基础上不断调用combine方法将初步结果进行聚合。
- CombinerAggregator的默认值会调用zero方法获得，与ReducerAggregator相比，它具有更好的并行度及效率。

CombinerAggregatorCombineImpl类是CombinerAggregator的执行器，它继承自Aggregator接口，其定义如下：

```
1 public class CombinerAggregatorCombineImpl implements Aggregator<Result> {
2     CombinerAggregator _agg;
3
4     public CombinerAggregatorCombineImpl(CombinerAggregator agg) {
5         _agg = agg;
6     }
7
8     public void prepare(Map conf, TridentOperationContext context) {
9     }
10
11    public Result init(Object batchId, TridentCollector collector) {
12        Result ret = new Result();
```

```
13          ret.obj = _agg.zero();
14          return ret;
15      }
16  
17      public void aggregate(Result val, TridentTuple tuple, TridentCollector collector) {
18          Object v = tuple.getValue(0);
19          if(val.obj==null) {
20              val.obj = v;
21          } else {
22              val.obj = _agg.combine(val.obj, v);
23          }
24      }
25  
26      public void complete(Result val, TridentCollector collector) {
27          collector.emit(new Values(val.obj));
28  }
```

- 第2行定义成员变量_agg，为CombinerAggregator类型。
- 第11~15行实现init方法，它调用_agg的zero方法来获得聚集的初始值。
- 第17~24行实现aggregate方法。参数val中含有目前聚集的结果，参数tuple中含有另外一个聚集的结果。于是在第18行，从第1列获得该聚集结果；第22行调用_agg的combine方法对目前聚集结果（val）以及另一个聚集结果（tuple）进行聚集。同样地，combine方法并不向外发送消息。
- 第26~28行将结果发送出去，目前结果只能有一列。
- 接口CombinerAggregator和ReducerAggregator通常会与ValueUpdater接口一起工作，以完成对存储的更新。

20.4 所有处理节点的上下文

Trident操作都是放在处理节点中执行的，接下来的两节内容将对处理节点进行分析讨论。处理节点会被放入Storm的Bolt节点中执行，每个Bolt节点可能含有多个处理节点。ProcessorContext类的Bolt中处理节点的执行的上下文环境，Bolt中的每一个处理节点可以利用ProcessorContext中state变量存储数据。ProcessorContext类的定义如下：

```
public class ProcessorContext {
    public Object batchId;
    public Object[] state;
    public ProcessorContext(Object batchId, Object[] state) {
        this.batchId = batchId;
        this.state = state;
    }
}
```

ProcessorContext对象中含有以下两个成员。

- batchId：事务序号。ITridentSpout对应于事务尝试消息（TransactionalAttemp），而IRichSpout中则为RichSpoutBatchId，内含一个随机数。

- state：对象数组，保存了该Bolt中每一个处理节点所对应的数据。理解该成员变量何时被赋值，以及何时使用是很关键的。

该对象在 SubTopologyBolt 类的 initBatchState 方法中完成初始化，相关代码如下：

```
@Override
public Object initBatchState(String batchGroup, Object batchId) {
    ProcessorContext ret = new ProcessorContext(batchId, new Object[_nodes.size()]);
    for(TridentProcessor p: _myTopologicallyOrdered.get(batchGroup)) {
        p.startBatch(ret);
    }
    return ret;
}
```

SubTopologyBolt中含有一系列的处理节点，它为每一个TridentProcessor均对应了一个对象以存储数据，这个对象即为ProcessorContext的对象数组state中的一个。对象数组state的大小与SubTopologyBolt中含有节点的数目相同。

每个处理节点的startBatch方法负责将上下文对象传入，但并不是每一个处理节点都需要存储数据。另外，该对象也实现了SubTopologyBolt内部处理节点之间的数据共享。

SubTopologyBolt是Trident中基本的Bolt类型，详情参见第26章。

20.4.1　单个处理节点的上下文

TridentContext对应于执行TrdientProcessor的上下文环境，它含有该处理节点输入输出所需要的信息，其定义如下：

```
public class TridentContext {
    Fields selfFields;
    List<Factory> parentFactories;
    List<String> parentStreams;
    List<TupleReceiver> receivers;
    String outStreamId;
    int stateIndex;
    BatchOutputCollector collector;
}
```

该类中各个成员的含义如下所示。

- selfFields为该处理节点新产生的列。
- parentFactories和parentStreams分别为该节点的父节点的消息模式和消息对应的流。
- receivers表明哪些处理节点将接收该节点产生的消息。
- outStreamId为该节点的输出消息流。
- stateIndex为该处理节点的编号，为SubTopologyBolt中的统一编号。
- collector用来向外发送消息。

20.4.2 操作执行的上下文

TrdientOperationContext为Trident的操作提供了上下文。它包含两个参数：_factory用来创建输入，_topoContext对象中含有Topology的上下文信息。相关代码如下：

```
public class TridentOperationContext implements IMetricsContext{
    TridentTuple.Factory _factory;
    TopologyContext _topoContext;
}
```

Trident的操作有时是需要了解Topology的上下文的（例如，当前Task的编号等）。

20.5 Trident 的输出收集器

TridentCollector接口主要用于向外发送处理节点的数据。由于一个处理节点的输出往往又是另外一个节点的输入，故其输出并不一定意味着真正地将消息发送出去，它可能会直接调用下一个处理节点的execute方法以完成当前节点的消息输出。通过这种方式，各个节点形成了一条调用链，并最终完成对消息的处理。

TridentCollector接口主要定义了emit方法和reportError方法，其定义如下：

```
public interface TridentCollector {
    void emit(List<Object> values);
    void reportError(Throwable t);
}
```


20.5.1 FreshCollector

FreshCollector类的emit方法会根据输入的values来创建消息，并调用该消息的接收端处理节点来执行该消息。FreshCollector的定义如下：

```
public class FreshCollector implements TridentCollector {
    FreshOutputFactory _factory;
    TridentContext _triContext;
    ProcessorContext context;

    public FreshCollector(TridentContext context) {
        _triContext = context;
        _factory = new FreshOutputFactory(context.getSelfOutputFields());
    }

    public void setContext(ProcessorContext pc) {
        this.context = pc;
    }

    @Override
    public void emit(List<Object> values) {
```

```
            TridentTuple toEmit = _factory.create(values);
            for(TupleReceiver r: _triContext.getReceivers()) {
                r.execute(context, _triContext.getOutStreamId(), toEmit);
            }
        }
    }
```

20.5.2　CaptureCollector

CaptuerCollector类的emit方法被调用时，只是将消息放在缓存中，并不真正发送出去。该类的定义如下：

```
public class CaptureCollector implements TridentCollector {
    public List<List<Object>> captured = new ArrayList();

    @Override
    public void emit(List<Object> values) {
        this.captured.add(values);
    }
}
```

20.5.3　GroupCollector

GroupCollector主要用于分组操作，负责缓存分组的消息。其中，emit方法传入的values为除了用于分组外的消息的其他部分。GroupCollector可以减少内存占用并提高效率。该类的定义如下：

```
public class GroupCollector implements TridentCollector {
    public List<Object> currGroup;

    ComboList.Factory _factory;
    TridentCollector _collector;

    public GroupCollector(TridentCollector collector, ComboList.Factory factory) {
        _factory = factory;
        _collector = collector;
    }

    @Override
    public void emit(List<Object> values) {
        List[] delegates = new List[2];
        delegates[0] = currGroup;
        delegates[1] = values;
        _collector.emit(_factory.create(delegates));
    }
}
```

注意，emit方法将缓存的currGroup与values合并后发送。

20.5.4 AppendCollector

对于输入的一条消息，AppendCollector类可能会产生多条消息，并且新产生的消息都要包含输入消息的所有列的情况。该类主要使用OperationOutputFactory对输入消息和产生的消息进行连接。AppendCollector类的定义如下：

```
1 public class AppendCollector implements TridentCollector {
2     OperationOutputFactory _factory;
3     TridentContext _triContext;
4     TridentTuple tuple;
5     ProcessorContext context;
6
7     public AppendCollector(TridentContext context) {
8         _triContext = context;
9         _factory = new OperationOutputFactory(context.getParentTupleFactories().get(0),
              context.getSelfOutputFields());
10    }
11
12    public void setContext(ProcessorContext pc, TridentTuple t) {
13        this.context = pc;
14        this.tuple = t;
15    }
16
17    @Override
18      public void emit(List<Object> values) {
19        TridentTuple toEmit = _factory.create((TridentTupleView) tuple, values);
20        for(TupleReceiver r: _triContext.getReceivers()) {
21            r.execute(context, _triContext.getOutStreamId(), toEmit);
22        }
23      }
24
25    @Override
26    public void reportError(Throwable t) {
27        _triContext.getDelegateCollector().reportError(t);
28    }
```

- 第4行的成员变量tuple表示产生的消息的前缀。
- 第18~23行在发送消息values时，首先利用_factory将成员tuple与输入消息values进行连接以得到新的消息，然后根据上下文对象获得该消息的接收处理节点，并最终调用这些节点的execute方法对输出消息进行处理。
- 第13~15行的setContext方法对tuple变量以及context进行设置。顾名思义，context为产生的消息的上下文，即消息的前缀。

20.5.5 AddIdCollector

Trident在进行数据处理时，消息中并不包含事务序号的信息，但在Storm中进行消息传输，却都是以事务序号作为第1列的列值的。AddIdCollector对发送消息的操作进行了简单的封装，当AddIdCollector发送消息时，事务序号被添加到第1列，之后消息才发送出去。AddIdCollector类的定义如下：

```
private static class AddIdCollector implements TridentCollector {
    BatchOutputCollector _delegate;
    Object _id;
    String _stream;

    public AddIdCollector(String stream, BatchOutputCollector c) {
        _delegate = c;
        _stream = stream;
    }


    public void setBatch(Object id) {
        _id = id;
    }

    @Override
    public void emit(List<Object> values) {
        _delegate.emit(_stream, new ConsList(_id, values));
    }

    @Override
    public void reportError(Throwable t) {
        _delegate.reportError(t);
    }
}
```

- AddIdCollector类实现了TridentCollector接口，并包含BatchOutputCollector的代理。_id表示为当前事务所对应的事务序号，_stream表示消息发送的目标流。emit方法中会新产生一个ConsList对象，它负责将_id与values连接起来。
- 与TridentTupleView的实现类似，这里并不会创建一个新的列表，而是重载AbstractList的get方法，在请求下标为0时返回txid，其他情况返回values中的数据。目前，这种TridentCollector只是被TridentSpoutExecutor使用。

20.6　Trident 的处理节点

接口TridentProcessor是Trident对操作执行的抽象，其类关系如图20-3所示。该接口是将Bolt节点中各个操作连接在一起的核心。

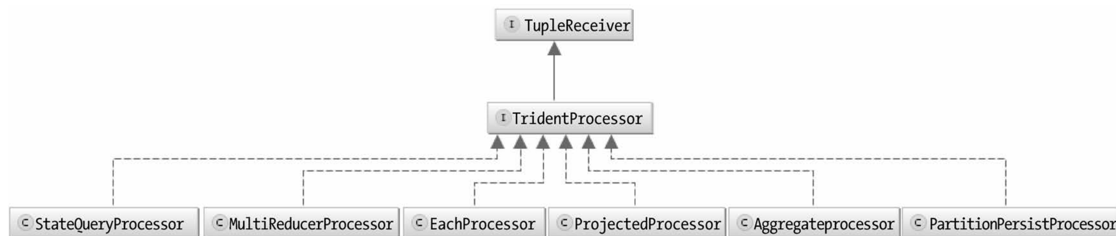


图20-3　TridentProcessor接口及其实现类

其中的基本接口以及基本实现类介绍如下。

- TupleReceiver：对基本的消息处理进行抽象。
- TridentProcessor：对基本的事务消息处理进行抽象。
- EachProcessor：会对消息进行逐条处理的处理节点。
- ProjectedProcessor：对映射消息进行操作的处理节点。
- PartitionPersistProcessor：分区存储的处理节点，它会将数据存储于State对象中。
- StateQueryProcessor：State查询处理节点，主要用于DRPC。
- MultiReducerProcessor：处理多流情况的TridentProcessor，详情请参见22.4节。

20.6.1 TridentProcessor接口

首先来看TupleReceiver接口的定义：

```
public interface TupleReceiver {
    void execute(ProcessorContext processorContext, String streamId, TridentTuple tuple);
}
```

该接口中定了execute方法，其输入为TridentTuple和该消息所对应的流号。Bolt中的各个操作，就是通过TupleReceiver的实例串接起来的（操作需属于同一个Bolt）。

TridentProcessor接口扩展了TupleReceiver接口，其定义如下：

```
public interface TridentProcessor extends Serializable, TupleReceiver {
    void prepare(Map conf, TopologyContext context, TridentContext tridentContext);
    void cleanup();
    void startBatch(ProcessorContext processorContext);
    void finishBatch(ProcessorContext processorContext);
    Factory getOutputFactory();
}
```

TridentProcessor是Trident对于事务处理操作的抽象。事务开始时，调用startBatch方法，并在处理属于一个事务的消息时，调用execute方法。事务处理结束后，则调用finishBatch方法。TridentProcessor的生命周期与Bolt相同，在Bolt被创建的时候调用prepare方法，Bolt销毁的时候调用cleanup方法。getOutputFactory方法用于返回该处理节点的输出模式。

接下来我们将介绍几种处理节点的重要实现。

20.6.2 PartitionPersistProcessor

PartitionPersistProcessor类会将一个事务中的数据以特定的方式写入到存储对象中，其定义如下：

```
public class PartitionPersistProcessor implements TridentProcessor {
    StateUpdater _updater;
    State _state;
    String _stateId;
```

```
    TridentContext _context;
    Fields _inputFields;
    ProjectionFactory _projection;
    FreshCollector _collector;
    public PartitionPersistProcessor(String stateId, Fields inputFields, StateUpdater updater) {
        _updater = updater;
        _stateId = stateId;
        _inputFields = inputFields;
    }
}
```

上述代码中，成员变量及构造函数的含义如下。

- State类型的_state对象为数据的存储目标。
- StateUpdater类型的_updater对象为数据的存储方法，即如何更新State对象。_stateId用来获取State对象。
- _inputFileds和_projection对象用于产生数据。

接下来讨论其主要的方法实现，相关代码如下：

```
1 public class PartitionPersistProcessor implements TridentProcessor {
2     @Override
3     public void prepare(Map conf, TopologyContext context, TridentContext tridentContext) {
4         List<Factory> parents = tridentContext.getParentTupleFactories();
5         if(parents.size()!=1) {
6             throw new RuntimeException("Partition persist operation can only have one parent");
7         }
8         _context = tridentContext;
9         _state = (State) context.getTaskData(_stateId);
10        _projection = new ProjectionFactory(parents.get(0), _inputFields);
11        _collector = new FreshCollector(tridentContext);
12        _updater.prepare(conf, new TridentOperationContext(context, _projection));
13    }
14
15    @Override
16    public void startBatch(ProcessorContext processorContext) {
17        processorContext.state[_context.getStateIndex()] = new ArrayList<TridentTuple>();
18    }
19
20    @Override
21    public void execute(ProcessorContext processorContext, String streamId, TridentTuple
          tuple) {
22        ((List) processorContext.state[_context.getStateIndex()]).add(_projection.create
                (tuple));
23    }
24
25    @Override
26    public void finishBatch(ProcessorContext processorContext) {
27        _collector.setContext(processorContext);
28        Object batchId = processorContext.batchId;
29        // since this processor type is a committer, this occurs in the commit phase
30        List<TridentTuple> buffer = (List) processorContext.state[_context.getStateIndex()];
31
```

```
32          // don't update unless there are tuples
33          // this helps out with things like global partition persist, where multiple tasks
                  may still
34          // exist for this processor. Only want the global one to do anything
35          // this is also a helpful optimization that state implementations don't need
                 to manually do
36          if(buffer.size() > 0) {
37              Long txid = null;
38              // this is to support things like persisting off of drpc stream,
                      which is inherently unreliable
39              // and won't have a tx attempt
40              if(batchId instanceof TransactionAttempt) {
41                  txid = ((TransactionAttempt) batchId).getTransactionId();
42              }
43              _state.beginCommit(txid);
44              _updater.updateState(_state, buffer, _collector);
45              _state.commit(txid);
46          }
47      }
48  }
```

- 第3~13行实现prepare方法。第9行根据_stateId获得存储对象，该对象是从Topology Context上下文对象中获得的。初始化SubTopologyBolt时，对state对象进行了集中的创建。第4~7行表示该处理节点不允许存在多个父节点，这是显然的。
- 第16~18行实现startBatch方法。ProcessorContext对象中的state成员用来存储局部数据，这里将其初始化为TridentTuple类型的列表。
- 第21~23行的execute方法实现中，仅仅是将输入消息放进了ProcessorContext对象的state列表中。
- 第26~47行在finishBatch方法中对state对象进行更新。即第44~45行分别调用state对象的beginCommit方法、_update对象的updateState方法以及_state对象的commit方法。第36行进行了检测，若无数据则不进行更新操作以提高效率。
- 目前的实现是较为简单的，更为直观的设计为：在事务开始时，调用beginCommit方法，在处理消息时，调用updateState方法，在事务结束时，调用commit方法。出于对性能的考虑，该实现为批处理模式，即在finishBatch中进行一次集中的更新。

20.6.3 StateQueryProcessor

与PartitionPersistProcessor相对应，StateQueryProcessor用于查询存储对象。由于它们都需要对存储对象进行操作，这两种TridentProcessor会被放在同一个Bolt节点中执行。首先来看该类的主要成员变量，代码如下：

```
public class StateQueryProcessor implements TridentProcessor {
    QueryFunction _function;
    State _state;
    String _stateId;
```

20

```
    TridentContext _context;
    Fields _inputFields;
    ProjectionFactory _projection;
    AppendCollector _collector;

    public StateQueryProcessor(String stateId, Fields inputFields, QueryFunctionfunction) {
        _stateId = stateId;
        _function = function;
        _inputFields = inputFields;
    }
}
```

下面来介绍各个成员变量的含义。

- ❑ QueryFunction类型的_function对象用于查询存储对象，而_stateId则用于在TopologyContext中获取_state对象。
- ❑ _inputFields和_projection对象用于构建_function的输入消息。_collector则用于收集输出。

下面讨论其主要的方法实现，代码如下：

```
1 public class StateQueryProcessor implements TridentProcessor {
2     @Override
3     public void prepare(Map conf, TopologyContext context, TridentContext tridentContext) {
4         List<Factory> parents = tridentContext.getParentTupleFactories();
5         if(parents.size()!=1) {
6             throw new RuntimeException("State query operation can only have one parent");
7         }
8         _context = tridentContext;
9         _state = (State) context.getTaskData(_stateId);
10        _projection = new ProjectionFactory(parents.get(0), _inputFields);
11        _collector = new AppendCollector(tridentContext);
12        _function.prepare(conf, new TridentOperationContext(context, _projection));
13    }
14
15    @Override
16    public void startBatch(ProcessorContext processorContext) {
17        processorContext.state[_context.getStateIndex()] =  new BatchState();
18    }
19
20    @Override
21    public void execute(ProcessorContext processorContext, String streamId, TridentTuple tuple) {
22        BatchState state = (BatchState) processorContext.state[_context.getStateIndex()];
23        state.tuples.add(tuple);
24        state.args.add(_projection.create(tuple));
25    }
26
27    @Override
28    public void finishBatch(ProcessorContext processorContext) {
29          BatchState state = (BatchState)processorContext.state[_context.getStateIndex()];
30        if(!state.tuples.isEmpty()) {
31            List<Object> results = _function.batchRetrieve(_state, state.args);
32            if(results.size()!=state.tuples.size()) {
33                throw new RuntimeException("Results size is different
                      than argumentsize: " + results.size() + " vs " + state.tuples.size());
```

```
34            }
35            for(int i=0; i<state.tuples.size(); i++) {
36                TridentTuple tuple = state.tuples.get(i);
37                Object result = results.get(i);
38                _collector.setContext(processorContext, tuple);
39                _function.execute(_projection.create(tuple), result, _collector);
40            }
41        }
42    }
43
44    private static class BatchState {
45        public List<TridentTuple> tuples = new ArrayList<TridentTuple>();
46        public List<TridentTuple> args = new ArrayList<TridentTuple>();
47    }
48 }
```

- 第3~13行实现prepare方法，它会根据stateId获取存储对象。
- 第44~47行为该处理节点在ProcessorContext中存储的数据结构，tuples表示输入的消息，而args则是输入消息中的某些列，代表了QueryFunction的查询参数。StateQueryProcessor在得到查询结果后，需要以某种方式将结果返回，于是在QueryFunction的execute方法中就需要传入原始的消息，即需要在存储中保存原始的消息。这样做是有意义的，例如在含有DRPC的Topology中，StateQueryProcessor收到的消息中可能含有两列:<查询消息来源，查询参数>，消息来源需要在所有处理节点中被保存，以使得最终结果可以返回给DRPC的服务器。
- 第16~18行的startBatch方法将processorContext中的state对象初始化为BatchState对象。
- 第21~25行的execute方法将输入的消息存入BatchState对象中，其中args是通过_projec tion对象的映射方法得到的。
- 第28~42行的finishBatch方法对state进行了查询。第31行调用QueryFunction对象的batch Retrieve方法对state对象进行查询。第32~34行对结果进行验证，即任何一条查询均需要有结果返回，并且是一对一的。第35~40行对每一条返回结果调用QueryFunction的execute方法，将结果发送到下一个节点或者存储。

20.7 聚集器的执行

有了前面关于处理节点的讨论后，本节来讨论聚集器是如何被Trident执行的这一问题。用户实现的Aggregator需要放置在处理节点中运行。

AggregateProcessor是其中的一个执行器，它继承自TridentProcessor接口，主要用来执行聚集器，其代码如下：

```
1 public class AggregateProcessor implements TridentProcessor {
2     Aggregator _agg;
3     TridentContext _context;
4     FreshCollector _collector;
```

```
5     Fields _inputFields;
6     ProjectionFactory _projection;
7
8     public AggregateProcessor(Fields inputFields, Aggregator agg) {
9         _agg = agg;
10        _inputFields = inputFields;
11    }
12
13    @Override
14    public void prepare(Map conf, TopologyContext context, TridentContext tridentContext) {
15        List<Factory> parents = tridentContext.getParentTupleFactories();
16        if(parents.size()!=1) {
17            throw new RuntimeException("Aggregate operation can only have one parent");
18        }
19        _context = tridentContext;
20        _collector = new FreshCollector(tridentContext);
21        _projection = new ProjectionFactory(parents.get(0), _inputFields);
22        _agg.prepare(conf, new TridentOperationContext(context, _projection));
23    }
24
25    @Override
26    public void cleanup() {
27        _agg.cleanup();
28    }
29
30    @Override
31    public void startBatch(ProcessorContext processorContext) {
32        _collector.setContext(processorContext);
33        processorContext.state[_context.getStateIndex()] = _agg.init
              (processorContext.batchId, _collector);
34    }
35
36    @Override
37    public void execute(ProcessorContext processorContext, String streamId,TridentTupletuple){
38        _collector.setContext(processorContext);
39        _agg.aggregate(processorContext.state[_context.getStateIndex()], _projection
              .create(tuple), _collector);
40    }
41
42    @Override
43    public void finishBatch(ProcessorContext processorContext) {
44        _collector.setContext(processorContext);
45        _agg.complete(processorContext.state[_context.getStateIndex()],_collector);
46    }
47
48    @Override
49    public Factory getOutputFactory() {
50        return _collector.getOutputFactory();
51    }
52 }
```

理解该类的重点在于分析聚集结果存储在什么地方以及如何在TridentProcessor的相应接口方法中调用聚集器的方法等。下面简要介绍上述代码的作用。

- 第22行，在TridentProcessor的prepare方法中调用聚集器的prepare方法，传入的参数为一个用于产生输入的工厂方法，以及处理节点的上下文对象。
- 第33行，在TridentProcessor的startBatch方法中，调用聚集器的init方法。完成对一个事务的初始化，初始化结果存储于所有处理节点的上下文对象中。
- 第39行，在TridentProcessor的execute方法中调用聚集器的aggregate方法进行聚集。传入的参数为当前聚集结果，以及要被聚集的消息；新的聚集结果将被存入处理节点的上下文对象中。
- 第45行，在TridentProcessor的finishBatch方法中，调用聚集器的complete方法。根据聚集器的具体实现，此处可能会将消息发送出去，故需传入一个输出收集器对象_collector。
- 在这些方法中，经常要设置当前的上下文对象，因为一个TridentProcessor可能会处理来自事务的消息（如含有DRPC查询的Topology）。

第 21 章

Trident流的基本操作

流是Trident的核心概念，Trident提供了关于流的多种操作。Trident将这些流的操作对应到了一张有向图上，通过添加节点和有向边来反映操作的变动。然后，Trident会根据这张有向图来对执行进行优化，并最终将其编译成为Topology，运行在Storm中。

对用户而言，Trident隐藏了基本的Spout和Bolt的概念。它通过流操作的概念来完成对逻辑的抽象，并最终将这些操作转换成为Spout或者Bolt节点。

本章将对单一流的基本操作进行介绍，下一章将进一步讨论多个流之间的操作。

21.1　流的成员变量和基础方法

本节介绍流的成员变量以及一些基础方法，这些是理解Trident流的基础。

21.1.1　流的成员变量

从其成员变量可以看出，流是与节点（Node）相对应的。Steam类的定义如下：

```
public class Stream implements IAggregatableStream {
    Node _node;
    TridentTopology _topology;
    String _name;
}
```

- 流中包含了一个Node类型的节点_node，以及流的名字_name。
- 同时，流中还包含了一个TridentTopology的引用_topology。TridentTopology中有一个有向图_graph，该图以流中_node节点为图顶点。关于节点类型Node，我们会在其他章节进行分析。

21.1.2　流节点名字

name函数会生成一个新的流，它根据前面介绍的_topology和_node两个成员变量来完成构建过程，相当于为当前_node节点赋予一个名字。由于Trident会将相关的处理节点放在一个Bolt中运行，节点的名字就很关键了，它可以用于获知哪些操作是在哪些Bolt节点上运行的。其代码如下：

```
public Stream name(String name) {
    return new Stream(_topology, name, _node);
}
```

下面的两个函数通过流的名字获取了Bolt节点的组件号，表现为在Storm UI上显示的Bolt名字。相关代码如下：

```
1 private static Map<Group, String> genBoltIds(Collection<Group> groups) {
2     Map<Group, String> ret = new HashMap();
3     int ctr = 0;
4     for(Group g: groups) {
5         if(!isSpoutGroup(g)) {
6             List<String> name = new ArrayList();
7             name.add("b");
8             name.add("" + ctr);
9             String groupName = getGroupName(g);
10            if(groupName!=null && !groupName.isEmpty()) {
11                name.add(getGroupName(g));
12            }
13            ret.put(g, Utils.join(name, "-"));
14            ctr++;
15        }
16    }
17    return ret;
18 }
19
20 private static String getGroupName(Group g) {
21    TreeMap<Integer, String> sortedNames = new TreeMap();
22    for(Node n: g.nodes) {
23        if(n.name!=null) {
24            sortedNames.put(n.creationIndex, n.name);
25        }
26    }
27    List<String> names = new ArrayList<String>();
28    String prevName = null;
29    for(String n: sortedNames.values()) {
30        if(prevName==null || !n.equals(prevName)) {
31            prevName = n;
32            names.add(n);
33        }
34    }
35    return Utils.join(names, "-");
36 }
```

- 第20行定义getGroupName函数，输入为一个节点组g。
- 第21~27行按照创建序号（creationIndex）对节点的名字进行排序，创建序号基本上与节点拓扑排序的顺序的是一致的，它反映了节点处理的先后顺序。
- 第27~35行将节点名字用“-”连接起来，并进行简单的前后去重。通常，使每个节点均拥有一个独特的名字会更利于调试，默认的情况下名字为空。
- 第1行定义的getBoltIds方法用于获得每一个节点组所对应的Bolt节点名（节点的ID），默认的模式为：b-[标号]-[getGroupname]。当getGroupName为null时，节点的ID只为b-[标号]。

21

目前，Spout节点的名字只能为spout-[标号]，如下面的函数所示：

```
private static Map<Node, String> genSpoutIds(Collection<SpoutNode> spoutNodes) {
    Map<Node, String> ret = new HashMap();
    int ctr = 0;
    for(SpoutNode n: spoutNodes) {
        ret.put(n, "spout" + ctr);
        ctr++;
    }
    return ret;
}
```

21.1.3　流的映射检查

Trident是以当前节点为出发点，根据相应的操作来创建新节点的，newStream和newDRPCStream用来创建最原始节点。

projectionValidation函数用来检查由当前节点创建的具有目标输出列的新节点是否合法，即需要projFields中的所有字段名都出现在当前节点的输出列中。该函数的定义如下：

```
private void projectionValidation(Fields projFields) {
    if (projFields == null) {
        return;
    }

    Fields allFields = this.getOutputFields();
    for (String field : projFields) {
        if (!allFields.contains(field)) {
            throw new IllegalArgumentException("Trying to select non-existent field:
                '" + field + "' from stream containing fields fields: <" + allFields + ">");
        }
    }
}
```

许多流操作都需要进行projectionValidation这项检查。

21.1.4　添加节点

addSourcedNoed函数在TridentTopology类中定义，主要被流和分组流中的方法调用。分组流是一种特殊的流，其中的消息会按照某些域进行分组。该函数的定义如下：

```
1 protected Stream addSourcedNode(Stream source, Node newNode) {
2     return addSourcedNode(Arrays.asList(source), newNode);
3 }
4 protected Stream addSourcedNode(List<Stream> sources, Node newNode) {
5     registerSourcedNode(sources, newNode);
6     return new Stream(this, newNode.name, newNode);
7 }
8
```

```
9 protected void registerSourcedNode(List<Stream> sources, Node newNode) {
10    registerNode(newNode);
11    int streamIndex = 0;
12    for(Stream s: sources) {
13        _graph.addEdge(s._node, newNode, new IndexedEdge(s._node, newNode, streamIndex));
14        streamIndex++;
15    }
16   }
17
18 protected void registerNode(Node n) {
19     _graph.addVertex(n);
20     if(n.stateInfo!=null) {
21         String id = n.stateInfo.id;
22         if(!_colocate.containsKey(id)) {
23             _colocate.put(id, new ArrayList());
24         }
25         _colocate.get(id).add(n);
26     }
27 }
```

- 第9~16行定义的registerSourcedNode用来向图中添加新节点。
- 第12~15行为每一个sources中的流中节点建立一条到新节点的边。注意，sources代表了新节点的所有父节点，此处会依据streamIndex对这些父节点进行标号。第10行调用registerNode添加新节点，该方法在第18~27行中定义。除在第19行向图中添加一个顶点外，若节点中的stateInfo成员不为空，则将该节点放入与存储序号（StateId）相对应的哈希表_colocate中。_colocate变量将所有访问同一存储的节点关联在一起，并将他们放在一个Bolt节点中执行。
- 第4~7行定义addSoucedNode方法，除利用registerSourcedNode方法创建新节点外，还会利用新节点产生一个流并返回。用户会对返回的流作进一步操作，进而创建更多的子节点、更多的边，这样，对流的操作便构成了有向图。

21.2 流映射操作

流映射操作比较简单，用于过滤输入流中的某些域，从而产生一个新的流。

```
public Stream project(Fields keepFields) {
    projectionValidation(keepFields);
    return _topology.addSourcedNode(this, new ProcessorNode(_topology.getUniqueStreamId(),
       _name, keepFields, new Fields(), new ProjectedProcessor(keepFields)));
}
```

- 流的映射操作是根据当前的流及要保留的域keepFields来创建一个新流的，该流以一个ProcessorNode作为节点。
- ProcessorNode中含有一个ProjectedProcessor，用来对输入的消息进行过滤，从而得到需要的列。ProjectedProcessor为TridentProcessor类型，用来执行映射操作。

- 映射操作操作会在图中创建新的节点。

21.3　流的分组操作

流的分组（groupBy）操作是非常关键的，该操作返回一个分组流，相关代码如下：

```
public GroupedStream groupBy(Fields fields) {
    projectionValidation(fields);
    return new GroupedStream(this, fields);
}
```

groupBy操作会创建一个分组流，分组流中含有当前的流引用以及指定对哪些域进行分组并作为成员变量的信息。

groupBy操作并不会创建新的节点，且分组流上进行的操作都是以分组的列作为聚合单位的。

分组流是多流操作的核心，最终会通过添加分区节点的方式转换成为流。在Trident底层，将通过域分组方式（Fields Grouping）完成，域分组所采用的域即为分组流的域，这样具有相同分组域的消息就可以到达相同的节点。

21.4　流的逐行操作

each操作表示对输入的每行消息进行处理，通常表示一条消息的处理与其他消息无关，该操作的代码如下：

```
1 public Stream each(Fields inputFields, Function function, Fields functionFields) {
2     projectionValidation(inputFields);
3     return _topology.addSourcedNode(this,
4             new ProcessorNode(_topology.getUniqueStreamId(),
5                 _name,
6                 TridentUtils.fieldsConcat(getOutputFields(), functionFields),
7                 functionFields,
8                 new EachProcessor(inputFields, function)));
9 }
```

- 形参function为Function类型，其输入的字段名由inputFields定义，functionFields表示新输出的字段名。
- 逐行操作将创建一个新的节点，节点的输出列为父节点的输出列加上函数新产生的列functionFields。
- 用户通常会实现Function接口，并利用each操作对流中的每条消息进行处理。
- EachProcessor用来调用输入的函数function。

21.5　流的分区操作

流的分区操作是非常关键的，它对应于Storm中的消息分组方式。在Trident中，消息的分组

方式是通过添加分区节点来完成的。与处理节点不同，分区节点只用来反映处理节点之间的消息接收方式，并不对应于实际的Spout或者Bolt节点。

下面定义的函数都与创建分区节点有关，讨论如下：

```
1 public Stream partitionBy(Fields fields) {
2     projectionValidation(fields);
3     return partition(Grouping.fields(fields.toList()));
4 }
5 public Stream partition(CustomStreamGrouping partitioner) {
6     return partition(Grouping.custom_serialized(Utils.serialize(partitioner)));
7 }
8
9 public Stream partition(Grouping grouping) {
10    if(_node instanceof PartitionNode) {
11        return each(new Fields(), new TrueFilter()).partition(grouping);
12    } else {
13        return _topology.addSourcedNode(this, new PartitionNode(_node.streamId, _name,
              getOutputFields(), grouping));
14    }
15 }
16
17 public Stream shuffle() {
18     return partition(Grouping.shuffle(new NullStruct()));
19 }
20
21 public Stream global() {
22     // use this instead of storm's built in one so that we can specify a
          singleemitbatchtopartition
23     // without knowledge of storm's internals
24     return partition(new GlobalGrouping());
25 }
26
27   public Stream batchGlobal() {
28       // the first field is the batch id
29       return partition(new IndexHashGrouping(0));
30 }
31 public Stream broadcast() {
32       return partition(Grouping.all(new NullStruct()));
33   }
34
35 public Stream identityPartition() {
36     return partition(new IdentityGrouping());
37 }
```

- 第9~15行是流分区操作的具体实现，它以Thrift类型的分组类型Grouping为形参。
- 如果当前节点已经为分区节点，它首先利用each操作添加一个TrueFilter节点，然后调用each操作产生的流的分区操作来添加分区节点，即第13行完成了一个分区节点的添加。
- 这样便保证了在有向图中没有任何两个分区节点是直接相连的。分区节点是用来创建节点组的分割节点，两个分区节点直接相连是没有意义的。

表21-1对分区操作的类型进行了总结。

表21-1　分区操作类型

函　　数	参　　数	意　　义
identityPartition()	new IdentityGrouping()	等同分区操作，分区前后Task的并行度相同
broadcast()	Grouping.all(new NullStruct())	全分组方式，原节点的消息将到达每一个目标节点
batchGlobal()	new IndexHashGrouping(0)	类似于域分组方式（Fields Grouping），根据消息的第1列内容选择目标节点
global()	new GlobalGrouping()	全局分组（Global Grouping）操作，消息到达第1个目标节点
shuffle()	Grouping.shuffle(new NullStruct())	等概率地随机选择目标节点
partitionBy(Fields fields)	Grouping.fields(fields.toList())	域分组，根据消息中选定列的内容的哈希值选择目标节点

- Trident的流分区操作对底层的分组方式进行了抽象，并实现了几种用户自定义的分组方式，使得接口更加易懂易用。例如用groupBy代替了域分组方式等。
- 在Trident中，传输消息的第1列通常为事务序号。batchGlobal操作会根据事务序号所在列进行域分组，使得相同事务序号所对应的数据可以到达同一个节点，这对于连接等操作是非常重要的。

21.6　流的单聚集器聚集操作

分区上聚集操作利用输入的聚集器对一个分区中属于同一事务的所有消息进行聚集。通常可以首先在分区内部进行聚集得到局部聚集结果，然后在此基础上进行全局聚集，得到最终的结果。这部分操作是Trident中最为复杂的部分，相关代码如下：

```
public Stream partitionAggregate(Fields inputFields, Aggregator agg, Fields functionFields) {
    projectionValidation(inputFields);
    return _topology.addSourcedNode(this,
            new ProcessorNode(_topology.getUniqueStreamId(),
                _name,
                functionFields,
                functionFields,
                new AggregateProcessor(inputFields, agg)));
}
```

- partitionAggregate操作创建了新的处理节点，内含AggregateProcessor类型的TridentProcessor。
- 输入的Aggregator由AggregateProcessor来调用执行。
- inputFields为agg的输入列，新创建节点的输出列为AggregatorProcessor的输出列，即为functionFields，与each的操作不同，partitionAggregate并不保留输入列。

21.7 流的多聚集器聚集操作

Trident中支持对于同一组输入消息同时运行多个聚集器，最终每个聚集器的聚集结果将通过叉积的形式发送出去。这部分内容是较为复杂的。

21.7.1 ChainedAggregatorDeclarer

类ChainedAggregatorDeclarer用来声明聚集器的链表。

AggType定义了聚集的类型，如下所示：

```
private static enum AggType {
    PARTITION,
    FULL,
    FULL_COMBINE
}
```

下面简要介绍各个成员的含义。

- ❑ PARTITION表示在某个分区上进行的聚集。
- ❑ FULL表示在所有分区上的聚集。
- ❑ FULL_COMBINE表示在所有分区上的合并聚集操作。Trident对于此类型的聚集进行了优化，会首先执行单独分区上的聚集，然后在其分区聚集的结果上进行全局聚集，以达到较高的效率。

类ChainedAggregatorDeclarer中的很多方法都返回了this指针，这使得用户可以通过“.”操作符来构建聚集器链。

类AggSpec用来描述聚集链中的Aggregator，其中含有输入字段名inFields、输出字段名outFields以及聚集器agg，其定义如下：

```
private static class AggSpec {
    Fields inFields;
    Aggregator agg;
    Fields outFields;
}
```

首先介绍类ChainedAggregatorDeclarer的成员变量，其代码如下：

```
List<AggSpec> _aggs = new ArrayList<AggSpec>();
IAggregatableStream _stream;
AggType _type = null;
GlobalAggregationScheme _globalScheme;
```

其中各个成员变量的含义如下。

- ❑ _aggs存储聚集链中的聚集器。
- ❑ _stream表示这些聚集器将要处理的流，可以为基本流也可以为分组流。
- ❑ _type表示聚集的类型。
- ❑ _globalScheme在全局聚集的时候有效。

类ChainedAggregatorDeclarer的成员变量globalScheme为GlobalAggregationScheme类型，下面介绍该类型如下：

```
public interface GlobalAggregationScheme<S extends IAggregatableStream> {
    IAggregatableStream aggPartition(S stream); // how to partition for second stage of aggregation
    BatchToPartition singleEmitPartitioner(); // return null if it's not single emit
}
```

aggPartition方法根据输入的流产生新流，而singleEmitPartitioner则用来获得BatchToParition的实现。目前该接口有两种实现，分别对应于不同目的。

类BatchGlobalAggScheme实现了该接口，其aggParition方法会根据事务序号的哈希值对消息进行分组。singleEmitPartitioner为IndexHashBatchToPartition对象，该对象则根据事务序号来选择发送消息的节点。类BatchGlobalAggScheme的定义如下：

```
static class BatchGlobalAggScheme implements GlobalAggregationScheme<Stream> {

    @Override
    public IAggregatableStream aggPartition(Stream s) {
        return s.batchGlobal();
    }

    @Override
    public BatchToPartition singleEmitPartitioner() {
        return new IndexHashBatchToPartition();
    }
}
```

类GlobalAggScheme中的aggPartition采用流的全局分组方式，singleEmitPartitioner为GlobalBatchToParition对象，即由第一个Task发送消息。类GlobalAggScheme的定义如下：

```
static class GlobalAggScheme implements GlobalAggregationScheme<Stream> {

    @Override
    public IAggregatableStream aggPartition(Stream s) {
        return s.global();
    }

    @Override
    public BatchToPartition singleEmitPartitioner() {
        return new GlobalBatchToPartition();
    }
}
```

- BatchGlobalAggScheme对象主要被用于流对象的aggregate方法中，即拥有同样事务序号的消息可以到达同一个节点。
- GlobalAggScheme对象主要被用于流的persistentAggregate方法中，表示是全局的聚集操作，并将聚集结果写入到存储对象中。

注意GroupedStream类也实现了接口GlobalAggregationScheme，相关代码如下：

```
public IAggregatableStream aggPartition(GroupedStream s) {
    return new GroupedStream(s._stream.partitionBy(_groupFields),_groupFields);
}
```

- 其aggPartition方法的实现为：利用_groupFields添加一个新的分区节点，并据此构建一个新的分组流。
- singleEmitPartitioner返回空，表示其不需要处理空的集合。

21.7.2 分区上的局部聚集操作

partitionAggregate用于完成分区上的聚集操作，该方法返回了一个ChainedPartitionAggregatorDeclarer对象，表明用户可以继续在该流上进行其他聚集操作。该方法的代码如下：

```
1 public ChainedPartitionAggregatorDeclarer partitionAggregate(Fields inputFields,
     Aggregator agg, Fields functionFields) {
2     _type = AggType.PARTITION;
3     _aggs.add(new AggSpec(inputFields, agg, functionFields));
4     return this;
5 }
6
7 public ChainedPartitionAggregatorDeclarer partitionAggregate(Fields inputFields,
      CombinerAggregator agg, Fields functionFields) {
8     initCombiner(inputFields, agg, functionFields);
9     return partitionAggregate(functionFields, new CombinerAggregatorCombineImpl(agg),
          functionFields);
10 }
11
12 public ChainedPartitionAggregatorDeclarer partitionAggregate(Fields inputFields,
      ReducerAggregator agg, Fields functionFields) {
13    return partitionAggregate(inputFields, new ReducerAggregatorImpl(agg), functionFields);
14 }
15
16 private void initCombiner(Fields inputFields, CombinerAggregator agg, Fields
      functionFields) {
17    _stream = _stream.each(inputFields, new CombinerAggregatorInitImpl(agg), functionFields);
18 }
```

- 第1~5行添加一个聚集器，同时指定其_type为PARTITION类型。
- 第7~10行将输入的CombinerAggregator适配成为Aggregator类型，需要分成两步来完成该操作。
    - 第一步，在第16~18行定义的initCombiner方法中，通过当前流的each操作来添加一个节点，该节点用来调用CombinerAggregator的init方法。Trident利用CombinerAggregatorInitImpl类对agg进行适配，注意_stream变量被更新为了each节点所对应的流，这也表明接下来的操作是发生在each节点上的。
    - 第二步，调用CombinerAggregatorCombineImpl将Combiner适配成聚集器。

21.7.3 全局聚集操作

aggregate方法对输入的流进行全局聚集操作，其输入为局部聚集操作的结果，其返回值为

ChainedFullAggregatorDeclarer类型，表明用户可以进一步对流进行聚集操作，进而构成一个聚集器的链。该方法的代码如下：

```
1 public ChainedFullAggregatorDeclarer aggregate(Fields inputFields, Aggregator agg,
      Fields functionFields) {
2     return aggregate(inputFields, agg, functionFields, false);
3 }
4
5 private ChainedFullAggregatorDeclarer aggregate(Fields inputFields, Aggregator agg,
      Fields functionFields, boolean isCombiner) {
6     if(isCombiner) {
7         if(_type == null) {
8             _type = AggType.FULL_COMBINE;
9         }
10    } else {
11        _type = AggType.FULL;
12    }
13    _aggs.add(new AggSpec(inputFields, agg, functionFields));
14    return this;
15 }
16
17  public ChainedFullAggregatorDeclarer aggregate(Fields inputFields,
        CombinerAggregator agg, Fields functionFields) {
18     initCombiner(inputFields, agg, functionFields);
19     return aggregate(functionFields, new CombinerAggregatorCombineImpl(agg),
           functionFields, true);
20 }
21
22  public ChainedFullAggregatorDeclarer aggregate(Fields inputFields,
        ReducerAggregator agg, Fields functionFields) {
23      return aggregate(inputFields, new ReducerAggregatorImpl(agg), functionFields);
24  }
```

第5~15行添加全局的聚集器，根据输入的isCombiner方法来判断参数agg的类型，依此为对_type进行赋值，经过适配后agg本身不能区分它是AggregatorReducer或者为CombinerReducer。其他方法与局部聚集器类似，这里不再赘述。

最后来看其核心方法chainEnd的实现，它是全局聚集生成的执行计划，其代码如下：

```
1 public Stream chainEnd() {
2     Fields[] inputFields = new Fields[_aggs.size()];
3     Aggregator[] aggs = new Aggregator[_aggs.size()];
4     int[] outSizes = new int[_aggs.size()];
5     List<String> allOutFields = new ArrayList<String>();
6     Set<String> allInFields = new HashSet<String>();
7     for(int i=0; i<_aggs.size(); i++) {
8         AggSpec spec = _aggs.get(i);
9         Fields infields = spec.inFields;
10        if(infields==null) infields = new Fields();
11        Fields outfields = spec.outFields;
12        if(outfields==null) outfields = new Fields();
13
14        inputFields[i] = infields;
```

```
15         aggs[i] = spec.agg;
16         outSizes[i] = outfields.size();
17         allOutFields.addAll(outfields.toList());
18         allInFields.addAll(infields.toList());
19     }
20     if(new HashSet(allOutFields).size() != allOutFields.size()) {
21         throw new IllegalArgumentException("Output fields for chained
               aggregators must be distinct: " + allOutFields.toString());
22     }
23 
24     Fields inFields = new Fields(new ArrayList<String>(allInFields));
25     Fields outFields = new Fields(allOutFields);
26     Aggregator combined = new ChainedAggregatorImpl(aggs, inputFields, new
           ComboList.Factory(outSizes));
27 
28     if(_type!=AggType.FULL) {
29         _stream = _stream.partitionAggregate(inFields, combined, outFields);
30     }
31     if(_type!=AggType.PARTITION) {
32         _stream = _globalScheme.aggPartition(_stream);
33         BatchToPartition singleEmit = _globalScheme.singleEmitPartitioner();
34         Aggregator toAgg = combined;
35         if(singleEmit!=null) {
36             toAgg = new SingleEmitAggregator(combined, singleEmit);
37         }
38         // this assumes that inFields and outFields are the same for combineragg
39         // assumption also made above
40         _stream = _stream.partitionAggregate(inFields, toAgg, outFields);
41     }
42     return _stream.toStream();
43 }
```


该方法利用ChainedAggregatorImpl来执行ChainedAggregatorDeclarer中定义的聚集链aggs。

- 第2~25行主要为ChainedAggregatorImpl的实例化做准备，同时检查aggs中是否有同名的列（这是不允许的），在第26行得到一个聚集器combined。
- 第28~41行根据聚集类型产生聚集计划。
  - PARTITION：只有第28~30行的partitionAggregate方法被执行，该方法会添加一个处理节点，并执行局部聚集操作。
  - FULL：只有第31~41行的partitionAggregate被执行。
  - FULL_COMBINE：则两个部分的partitionAggregate都被执行，表示为先在分区上进行局部聚集，然后再在分区的结果上进一步聚集。
- 第32~37行用来实现全局的聚集操作。根据对_globalScheme对象的分析，若为流对象，需要处理集合为空的情况，则利用SingleEmitAggregator对聚集器combined进行包装；若为分组流对象则不需要进行这一操作。

21.7.4 含有多个聚集器的partitionAggregate操作

类ChainedAggregatorDeclarer可用来添加多个聚集器，该类的定义如下：

```
1 public ChainedAggregatorDeclarer chainedAgg() {
2     return new ChainedAggregatorDeclarer(this, new BatchGlobalAggScheme());
3 }
4
5 public Stream partitionAggregate(Fields inputFields, CombinerAggregator agg,
      Fields functionFields) {
6     projectionValidation(inputFields);
7     return chainedAgg()
8            .partitionAggregate(inputFields, agg, functionFields)
9            .chainEnd();
10}
11
12 public Stream partitionAggregate(Fields inputFields, ReducerAggregator
      agg,Fields functionFields) {
13    projectionValidation(inputFields);
14    return chainedAgg()
15           .partitionAggregate(inputFields, agg, functionFields)
16           .chainEnd();
17 }
```

- 第1~3行定义的chainedAgg方法用来获得ChainedAggregatorDeclarer对象，注意该对象操作的流为当前流，GlobalAggScheme对象为BatchGlobalAggSheme对象。
- 第5~10行添加一个CombinerAggregator聚集器，第12~17行添加一个ReducerAggregator聚集器， chainEnd方法会将这些聚集器统一启动。
- 可以看出，目前的partitionAggregate方法中只存在一个聚集器。若希望含有多个聚集器，则用户需要使用类似的方法调用chainedAgg来添加多个聚集器，最后再调用chainEnd方法。
- Trident这部分的设计过于复杂。

21.8 流的聚集操作

aggregate方法与partitionAggregate的实现基本一致，它代表了全局的聚集。它也对ReducerAggregator和CombinerAggregator进行了适配，这里不再过多讨论。aggregate方法的实现代码如下：

```
public Stream aggregate(Fields inputFields, Aggregator agg, Fields functionFields) {
    projectionValidation(inputFields);
    return chainedAgg()
           .aggregate(inputFields, agg, functionFields)
           .chainEnd();
}

public Stream aggregate(Fields inputFields, CombinerAggregator agg, Fields functionFields) {
    projectionValidation(inputFields);
    return chainedAgg()
           .aggregate(inputFields, agg, functionFields)
           .chainEnd();
}
```

```
public Stream aggregate(ReducerAggregator agg, Fields functionFields) {
    return aggregate(null, agg, functionFields);
}

public Stream aggregate(Fields inputFields, ReducerAggregator agg, Fields functionFields) {
    projectionValidation(inputFields);
    return chainedAgg()
            .aggregate(inputFields, agg, functionFields)
            .chainEnd();
}
```

21.9 流的分区写入操作

partitionPersist操作用于将一个分区的数据存储到对象中，该方法返回一个TridentState对象，用户可以对该对象执行查询操作（例如DRPC）。partitionPersist的代码如下：

```
1 public TridentState partitionPersist(StateFactory stateFactory,
      Fields inputFields, StateUpdater updater, Fields functionFields) {
2     return partitionPersist(new StateSpec(stateFactory), inputFields, updater, functionFields);
3 }
4
5 public TridentState partitionPersist(StateSpec stateSpec, Fields inputFields,
      StateUpdater updater, Fields functionFields) {
6     projectionValidation(inputFields);
7     String id = _topology.getUniqueStateId();
8     ProcessorNode n = new ProcessorNode(_topology.getUniqueStreamId(),
9         _name,
10        functionFields,
11        functionFields,
12        new PartitionPersistProcessor(id, inputFields, updater));
13    n.committer = true;
14    n.stateInfo = new NodeStateInfo(id, stateSpec);
15    return _topology.addSourcedStateNode(this, n);
16 }
17 }
```

- 第5~17行是该操作的主要实现，StateSpec类型的stateSpec对象用来帮助创建存储State对象，StateUpdater类型的updater对象用来对该State对象进行更新。
- 第6~12行创建了一个处理节点，该节点含有PartitionPersistProcessor对象，用于利用StateUpdater更新State对象的数据。
- 第14行节点的stateInfo对象被初始化。在声明SubTopology对象时，将根据stateInfo对象创建State对象。
- 第15行调用addSourcedStateNode方法创建一个节点。该节点与普通节点的区别在于，addSourcedStateNode方法返回了TridentState对象，该对象代表了操作的终止，所以不存在子节点。第2行利用stateFactory对象来构建StateSpec对象，是addSourcedStateNode方法的一个重载。

21.10　查询操作

查询操作并非属于流操作的一部分，但它可用来查询流写入的存储对象，相关代码如下：

```
1 public Stream stateQuery(TridentState state, Fields inputFields, QueryFunction
      function, Fields functionFields) {
2     projectionValidation(inputFields);
3     String stateId = state._node.stateInfo.id;
4     Node n = new ProcessorNode(_topology.getUniqueStreamId(),
5                     _name,
6                     TridentUtils.fieldsConcat(getOutputFields(), functionFields),
7                     functionFields,
8                     new StateQueryProcessor(stateId, inputFields, function));
9     _topology._colocate.get(stateId).add(n);
10    return _topology.addSourcedNode(this, n);
11 }
```

- 第4~8行创建一个处理节点，该节点中含有StateQueryProcessor，因此可用来查询State对象。
- 第9行将该节点放入_coloctate对象的映射中。键为stateId，值为对同一个存储对象进行操作的所有节点，在Topology执行优化的过程中，会将这些节点放入同一个Bolt中来执行。

21.11　流的全局写入操作

persistentAggreate操作即为在全局的节点上执行partitionPersist操作，其代码如下：

```
public TridentState persistentAggregate(StateSpec spec, Fields inputFields,
    ReducerAggregator agg, Fields functionFields) {
    projectionValidation(inputFields);
    return global().partitionPersist(spec, inputFields, new
        ReducerAggStateUpdater(agg), functionFields);
}
```

在上述代码中，我们首先利用global()函数创建分区节点，使得所有的数据都汇集到一个节点上，然后在该节点上调用partitionPersist方法进行聚集并存储。

21.12　流的操作与有向图构建

表21-2列举了流的各种方法与构建有向图的关系，并且给出了相关类。

表21-2　流操作与有向图构建

操作名称	节　点	相　关　类
project	创建处理节点	ProjectedProcessor
groupBy	不创建新的节点	返回一个GroupedStream对象
partitionBy	创建分区节点	Fields Grouping
each	创建处理节点	EachProcessor

（续）

操作名称	节　点	相　关　类
partitionAggregate	创建处理节点，根据需要创建分区节点	CombinerAggregatorCombineImpl CombinerAggregatorInitImpl ReducerAggregatorImpl ChainedAggregatorImpl ChainedAggregatorDeclarer BatchGlobalAggScheme
aggregate	创建处理节点，根据需要创建分区节点	CombinerAggregatorCombineImpl CombinerAggregatorInitImpl ReducerAggregatorImpl ChainedAggregatorImpl ChainedAggregatorDeclarer BatchGlobalAggScheme
partitoinPersist	创建处理节点	ReducerAggStateUpdater
stateQuery	创建处理节点	StateQueryProcessor
persistentAggregate	创建分区节点以及处理节点	Global Grouping GlobalAggregationScheme

21.13　分组流

当流进行分组操作时，将会产生分组流对象。接下来的操作都是在利用分组域做键进行分组后的消息集合上进行的，这与SQL的设计相类似。分组流中的操作经常需要借助分区节点帮助，以及借助在分组内消息集合上聚集操作的协助。学习这个类的时候可以对比基本流的实现，并思考分组流与基本流是如何互相转换的。

21

21.13.1　成员变量

GroupedStream类的定义如下所示：

```
public class GroupedStream implements IAggregatableStream, GlobalAggregationScheme<GroupedStream> {
    Fields _groupFields;
    Stream _stream;
}
```

其中各个成员变量的含义如下。

- _groupFields表示用于进行分组的域。
- _stream为基本流对象。

21.13.2　逐行操作

分组流上的each操作将通过调用基本流的each操作来完成，这个操作将继续返回分组流。由于each操作是在消息层面进行处理而不需要聚集，所以分组流的each操作与基本流的类似。其代码如下：

```
public IAggregatableStream each(Fields inputFields, Function function, Fields functionFields) {
    Stream s = _stream.each(inputFields, function, functionFields);
    return new GroupedStream(s, _groupFields);
}
```

21.13.3　分组流的分区聚集操作

分组流的partitionAggregate操作是利用GroupedAggregator来完成的。该聚集器会首先调用_groupFields对输入消息进行分组，然后在分组后的集合上调用聚集器agg，该操作将返回一个分组流。相关代码如下：

```
public IAggregatableStream partitionAggregate(Fields inputFields, Aggregator agg,
    Fields functionFields) {
    Aggregator groupedAgg = new GroupedAggregator(agg, _groupFields, inputFields, function
        Fields.size());
    Fields allInFields = TridentUtils.fieldsUnion(_groupFields, inputFields);
    Fields allOutFields = TridentUtils.fieldsConcat(_groupFields, functionFields);
    Stream s = _stream.partitionAggregate(allInFields, groupedAgg, allOutFields);
    return new GroupedStream(s, _groupFields);
}
```

21.13.4　查询操作

stateQuery操作首先调用流的分区操作，其输入参数为_groupFields。这样就创建了域分区节点，随着数据在不同的Bolt节点之间传输，_groupFields值相同的节点将到达相同的目标节点。partitionBy操作返回基本流对象，然后这里将调用基本流对象的stateQuery操作。

分组流的stateQuery操作返回基本流对象，其代码如下：

```
public Stream stateQuery(TridentState state, Fields inputFields, QueryFunction function,
    Fields functionFields) {
    return _stream.partitionBy(_groupFields)
                  .stateQuery(state,
                     inputFields,
                     function,
                     functionFields);
}
```

21.13.5　聚集操作

分组流上aggregate操作的实现较为复杂，相关代码如下：

```
1 public ChainedAggregatorDeclarer chainedAgg() {
2     return new ChainedAggregatorDeclarer(this, this);
3 }
4
5 public Stream aggregate(Fields inputFields, Aggregator agg, Fields functionFields) {
6     return new ChainedAggregatorDeclarer(this, this)
7             .aggregate(inputFields, agg, functionFields)
8             .chainEnd();
```

```
9 }
10
11 public IAggregatableStream aggPartition(GroupedStream s) {
12     return new GroupedStream(s._stream.partitionBy(_groupFields), _groupFields);
13 }
```

- 第1~3行定义了chainedAgg方法，注意ChainedAggregatorDeclarer的第二个参数传入了this指针，且分组流也实现了接口GlobalAggregationScheme，所以可以这样调用。
- 于是，在ChainedAggregatorDeclarer的chainEnd方法中将调用定义在第11~13行的aggPartition方法，该方法对源基本流调用分区操作，从而创建新的分区节点。

摘抄chainEnd函数的部分代码如下：

```
1 public Stream chainEnd() {
2
3     Aggregator combined = new ChainedAggregatorImpl(aggs, inputFields, new ComboList.
          Factory(outSizes));
4
5     if(_type!=AggType.FULL) {
6         _stream = _stream.partitionAggregate(inFields, combined, outFields);
7     }
8     if(_type!=AggType.PARTITION) {
9         _stream = _globalScheme.aggPartition(_stream);
10        BatchToPartition singleEmit = _globalScheme.singleEmitPartitioner();
11        Aggregator toAgg = combined;
12        if(singleEmit!=null) {
13            toAgg = new SingleEmitAggregator(combined, singleEmit);
14        }
15         // this assumes that inFields and outFields are the same for combineragg
16        // assumption also made above
17        _stream = _stream.partitionAggregate(inFields, toAgg, outFields);
18    }
19    return _stream.toStream();
20 }
```

- 第9行_stream对象为分组流，它含有groupFields进行分区的节点。
- 由于第12行的singleEmit为空，所以将调用分组流上的partitionAggregate方法（第17行）。
- 第19行调用_stream.toStream方法返回分组流中的基本流对象。
- 该操作的含义是：首先按照_groupedFields对输入流进行重新分区，然后在每个分区上调用基于分组的聚集算法。由于具有相同_groupedFields值的消息都已经在同一个节点上了，所以返回的将是流对象而不是分组流对象。

21.13.6 写入操作

persistentAggregate操作是基于aggregate操作来完成的，其代码如下：

```
public TridentState persistentAggregate(StateSpec spec, Fields inputFields, CombinerAggregator
    agg, Fields functionFields) {
    return aggregate(inputFields, agg, functionFields)
```

```
        .partitionPersist(spec,
            TridentUtils.fieldsUnion(_groupFields, functionFields),
                new MapCombinerAggStateUpdater(agg, _groupFields, functionFields),
                TridentUtils.fieldsConcat(_groupFields, functionFields));
}
```

- 根据前面的分析，分组流上的aggregate操作将返回已经按照_groupFields进行分区且聚集的流对象。然后，调用流对象的partitionPersist方法。
- 利用MapCombinerAggStateUpdater来对已根据_groupFields进行分组的存储数据进行更新，其State对象为MapState，键为_groupFields，值为_groupFields上分组集合的聚集结果。

21.14　利用流操作来构建 Topology 的例子

在Trident中，用户通过流的操作来完成Topology的构建，Trident则需要将用户的代码合理地分配到Storm的运行节点中。Trident进行了较好的抽象，使用户可以更加专注于代码逻辑。但这也带来了额外的复杂性，即当出现问题时，用户需要清楚自己代码的运行和部署过程。

下面的代码来自于StormStarter项目中关于Trident的一个例子，这部分代码十分精炼，本节将分析下面的代码如何被构建成为Topology。

```
1 TridentState wordCounts =
2       topology.newStream("spout1", spout)
3         .parallelismHint(16)
4         .each(new Fields("sentence"), new Split(), new Fields("word"))
5         .groupBy(new Fields("word"))
6         .persistentAggregate(new MemoryMapState.Factory(),
7                              new Count(), new Fields("count"))
8         .parallelismHint(16);
```

- 第2行调用newStream，将产生一个id为s1的流，该流含有一个Spout节点。为便于理解，下面我们列举该节点在运行时成员变量的值。

```
storm.trident.planner.SpoutNode[
    spout=storm.trident.testing.FixedBatchSpout
    txId=spout1
    type=BATCH
    name=<null>
    allOutputFields=[sentence]
    streamId=s1
    creationIndex=1]
```

- 第3行为该节点设置并行度。
- 第4行的each操作将会产生一个新的流s2。它将s1作为输入流，构建了一条从s1到s2的有向边。
- 第5行将产生一个分组流，分组流对象包含两个成员变量。一个是_groupedFields，指明按照哪些Fields进行分组，在我们的例子中为word列。另一个是_stream，将要进行分组的流对象，在我们的例子中为流s2。

- 第6行将对分组后的流进行聚集操作，也是最为复杂的部分。大体上经过三个步骤：
  - 在每个分区上进行局部的聚集操作；
  - 然后在单一节点上在对局部聚集的结果进行聚集，这样我们可以得到全局的聚集结果；
  - 最后将全局的结果存储在State对象中。

首先看一下CombinerAggregator接口，其代码如下：

```
public interface CombinerAggregator<T> extends Serializable {
    T init(TridentTuple tuple);
    T combine(T val1, T val2);
    T zero();
}
```

它主要包含3个成员方法，init方法将输入的Tuple转化成为对象T，combine方法将对输入的两个T对象进行操作并产生新的对象T，zero方法用来表示默认值。通过这些接口方法，Trident可以构建起一颗层级的树状结构以完成最终的聚集操作。

接下来我们来看一下Count这个CombinerAggregator的实现，其代码如下：

```
public class Count implements CombinerAggregator<Long> {
    @Override
    public Long init(TridentTuple tuple) {
        return 1L;
    }
    @Override
    public Long combine(Long val1, Long val2) {
        return val1 + val2;
    }
    @Override
    public Long zero() {
        return 0L;
    }
}
```

它的目标是分布式地计算输入消息的行数，于是在init方法中返回1，在combine方法中返回输入的两个临时聚集的和，在zero方法中返回0。

例子中第6~7行用来将Count应用在节点中，以分布式地完成计算。其中，我们需要对每一个输入的消息调用Count的init方法，在Trident中利用EachProcessor来完成这项工作，在EachProcessor中将调用Count的init方法。于是Trident创建了流s3，其基本信息如下：

```
storm.trident.planner.ProcessorNode [
    committer=false
    processor=storm.trident.planner.processor.EachProcessor
    selfOutFields=[count]
    nodeId=3133ab19-17da-4d74-a9c2-a4f022406560
    name=<null>
    allOutputFields=[sentence, word, count]
    streamId=s3
    creationIndex=3
]
```

在EachProcessor中将调用CombinerAggregatorInitImpl，然后CombinerAggregatorInitImpl将调用Count的init方法。该流的输出为[sentence, word, count]，显然count为0。CombinerAggregatorInitImpl的代码如下：

```
public class CombinerAggregatorInitImpl implements Function {

    CombinerAggregator _agg;
    public CombinerAggregatorInitImpl(CombinerAggregator agg) {
        _agg = agg;
    }
    @Override
    public void execute(TridentTuple tuple, TridentCollector collector) {
        collector.emit(new Values(_agg.init(tuple)));
    }
    @Override
    public void prepare(Map conf, TridentOperationContext context) {
    }
    @Override
    public void cleanup() {
    }
}
```

在这个简单的包装类中，execute方法内部会调用内嵌的_agg.init(tuple)，并将结果进行输出。其中Fucntion接口是比较基础的，它表示对输入的消息进行处理，并通过collector将其输出，其代码如下：

```
public interface Function extends EachOperation {
    void execute(TridentTuple tuple, TridentCollector collector);
}
```

接下来，Trident需要对已经初始化的消息进行聚集操作。因为是在分组后的流上进行的，所以所有的聚集操作都是以分组域作为关键字的，即表示我们将对单词进行计数，而不是整个句子进行计数。基本上具有如下调用嵌套关系：

(1) storm.trident.planner.processor.AggregateProcessor

(2) storm.trident.operation.impl.GroupedAggregator

(3) storm.trident.operation.impl.ChainedAggregatorImpl

(4) storm.trident.operation.impl.CombinerAggregatorCombineImpl

(5) storm.trident.operation.builtin.Count

- 在4的aggregate方法中，将调用Count的combine方法。
- 3可以包含多个类似于4的aggregate，并依次分别执行。
- 2将在分组好的消息的基础上执行3。
- 1中将调用2的aggregate方法。

这些操作被封装在一个新的流s4中，其基本信息如下：

```
storm.trident.planner.ProcessorNode
    committer=false
    processor=storm.trident.planner.processor.AggregateProcessor
    selfOutFields=[word, count]
    name=<null>
    allOutputFields=[word, count]
    streamId=s4
    parallelismHint=<null>
    creationIndex=4
]
```

接下来，Trident会在初步聚集的节点后面插入一个含有实际聚集操作的节点，该节点主要包含一个分组的域。

```
storm.trident.planner.PartitionNode@46d8694[
    name=<null>
    allOutputFields=[word, count]
    streamId=s4
    parallelismHint=<null>
    stateInfo=<null>
    creationIndex=5
]
```

注意，其`streamId`为s4，但是`creationIndex`为5，表示其为一个新的流，该节点会导致数据从初步聚集的节点发送到一个单一的全局聚集节点，所以用s4作为`StreamId`也是可以的。

然后，Trident会在全局节点上运行与流s4上非常相近的操作，只是不需要再次调用`init`方法或再次产生一个新的节点了。流的基本信息如下：


```
storm.trident.planner.ProcessorNode
    committer=false
    processor=storm.trident.planner.processor.AggregateProcessor
    selfOutFields=[word, count]
    nodeId=6aa7a808-1627-4871-a8a9-4151148ef2b0
    name=<null>
    allOutputFields=[word, count]
    streamId=s5
    parallelismHint=<null>
    stateInfo=<null>
    creationIndex=6
]
```

第 22 章 Trident中流的交互操作

Trident中流的交互操作涉及多个流之间的交互，如流的连接或合并等。基本的类关系如图22-1所示。

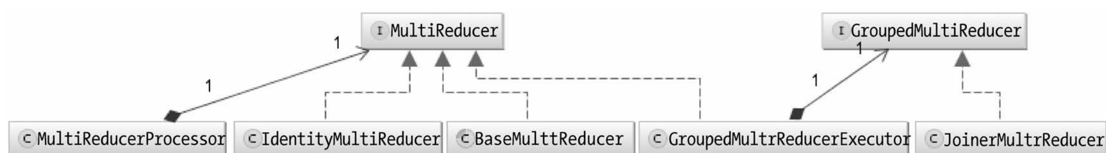


图22-1　多流操作的类关系

由于每一个流都是由无限的消息构成的，Trident中的多流操作仅局限于某一个事务内部。例如，用户要进行Stream$_1$和Stream$_2$关于第1列的连接操作：

在txId=1时，Stream$_1$发送消息<A_1, 1>，<A_2,2>；

在txId=2时，Stream$_1$发送消息<A_1, 3>，<A_2, 4>；

在txId=1时，Stream$_2$发送消息<A_1, 5>，<A_2,6>；

在txId=2时，Stream$_2$发送消息<A_1, 7>，<A_2,8>。

即Stream$_1$和Stream$_2$分别发送出去4条消息。然而在进行连接操作时，只能在一个事务内部进行，事务之间不能进行连接，则得到如下结果：

txId=1时，<A_1, 1, 5>，<A_2, 2, 6>；

txId=2时，<A_1, 3, 7>，<A_2, 4, 8>。

若想进行全局的连接操作，用户需要将其中一个流的结果存储为State对象，然后让另外一个流通过StateQuery操作来查询第一个流的结果。

22.1　基本接口

MultiReducer接口主要用于对多个流进行操作。Trident利用MultiReducerProcessor来执行MultiReducer，其execute方法传入的形参streamIndex代表了输入的消息是从哪个流获得的。接口的代码如下：

```
public interface MultiReducer<T> extends Serializable {
    void prepare(Map conf, TridentMultiReducerContext context);
    T init(TridentCollector collector);
    void execute(T state, int streamIndex, TridentTuple input, TridentCollector collector);
    void complete(T state, TridentCollector collector);
    void cleanup();
}
```

而GroupedMultiReducer接口中execute方法所收到的消息，除了带有streamIndex信息之外，都属于同一个特定分组，所以GroupedMultiReducer主要用作连接操作。该接口的实现如下：

```
public interface GroupedMultiReducer<T> extends Serializable {
    void prepare(Map conf, TridentMultiReducerContext context);
    T init(TridentCollector collector, TridentTuple group);
    void execute(T state, int streamIndex, TridentTuple group, TridentTuple input, TridentCollector
       collector);
    void complete(T state, TridentTuple group, TridentCollector collector);
    void cleanup();
}
```

22.2 JoinerMultiReducer

JoinState结构是JoinerMultiReducer进行连接的数据结构。其中，sides存储了每个流已经收到的消息，numSidesReceived表示已经从哪些流收到了消息，group表示进行连接的值，indices表示sides中元素的下标。JoinState的代码如下：


```
public static class JoinState {
    List<List>[] sides;
    int numSidesReceived = 0;
    int[] indices;
    TridentTuple group;

    public JoinState(int numSides, TridentTuple group) {
        sides = new List[numSides];
        indices = new int[numSides];
        this.group = group;
        for(int i=0; i<numSides; i++) {
            sides[i] = new ArrayList<List>();
        }
    }
}
```

JoinerMultiReducer类用于完成多个流的连接操作。

22.2.1 成员变量及构造函数

首先分析JoinerMultiReducer所包含的数据和构造方法，相关代码如下：

```
1 public class JoinerMultiReducer implements GroupedMultiReducer<JoinState> {
2
3     List<JoinType> _types;
4     List<Fields> _sideFields;
5     int _numGroupFields;
6     ComboList.Factory _factory;
7
8
9     public JoinerMultiReducer(List<JoinType> types, int numGroupFields, List<Fields> sides) {
10        _types = types;
11        _sideFields = sides;
12        _numGroupFields = numGroupFields;
13    }
14    public void prepare(Map conf, TridentMultiReducerContext context) {
15        int[] sizes = new int[_sideFields.size() + 1];
16        sizes[0] = _numGroupFields;
17        for(int i=0; i<_sideFields.size(); i++) {
18            sizes[i+1] = _sideFields.get(i).size();
19        }
20        _factory = new ComboList.Factory(sizes);
21    }
22 }
```

_types为连接类型，其定义如下：

```
public enum JoinType {
    INNER,
    OUTER;
}
```

- 该类型反映了某一个输入流在没有消息时该如何进行处理。若为OUTER类型，将用List<null>来表示该流的消息；若为INNER类型且没有消息，则整个连接将没有输出。在Trident中，_types是流的属性，即_types的数目与进行连接的流的数目相同。
- _sideFields是每一个参加连接的流的模式。
- _numGroupFields为连接的键的数目。
- _factory用来产生最终的输出，输出的列数为_numGroupFields加上_sideFields中每一个流的列数。
- 从第14~21行定义的prepare方法可以看出，_factory共有N+1部分构成，其中N为参与连接的流的个数。

init函数根据分组域构建JoinState对象，消息group为连接的连接键，它是各个流的列中相同的部分。函数init的代码如下：

```
public JoinState init(TridentCollector collector, TridentTuple group) {
    return new JoinState(_types.size(), group);
}
```

22.2.2 execute方法

Trident保证JoinerMultiReducer收到的消息都属于同一个分组，即在exeucte方法中形参group的值与init方法中形参group的值相同。也就是说，execute方法收到的消息都是满足连接条件的，理解这点很重要。execute方法完成了内连接（InnerJoin），其代码如下：

```
1 public void execute(JoinState state, int streamIndex, TridentTuple group, TridentTuple input,
      TridentCollector collector) {
2      //TODO: do the inner join incrementally, emitting the cross join with this tuple, against
           all other sides
3     //TODO: only do cross join if at least one tuple in each side
4     List<List> side = state.sides[streamIndex];
5     if(side.isEmpty()) {
6         state.numSidesReceived++;
7     }
8
9     side.add(input);
10    if(state.numSidesReceived == state.sides.length) {
11        emitCrossJoin(state, collector, streamIndex, input);
12    }
13}
```

- 第4行与第9行根据streamIndex找到JoinState对象中用于存储该流消息的列表，然后将该消息存储。
- 第5~7行，若第一次从某流收到消息，则更新numSidesReceived变量。
- 第10行，若已经从参与连接的流都收到了消息，则已经可以进行连接了。接下来，调用emitCrossJoin将连接上的消息发送出去，这种递增地进行内连接的方法是算法的核心。

下面来看emitCrossJoin的实现，其代码如下：

```
1 private void emitCrossJoin(JoinState state, TridentCollector collector, int overrideIndex,
      ridentTuple overrideTuple) {
2     List<List>[] sides = state.sides;
3     int[] indices = state.indices;
4     for(int i=0; i<indices.length; i++) {
5         indices[i] = 0;
6     }
7
8     boolean keepGoing = true;
9     //emit cross-join of all emitted tuples
10    while(keepGoing) {
11        List[] combined = new List[sides.length+1];
12        combined[0] = state.group;
13        for(int i=0; i<sides.length; i++) {
14            if(i==overrideIndex) {
15                combined[i+1] = overrideTuple;
16            } else {
17                combined[i+1] = sides[i].get(indices[i]);
18            }
19        }
```

```
20          collector.emit(_factory.create(combined));
21          keepGoing = increment(sides, indices, indices.length - 1, overrideIndex);
22      }
23  }
24
25
26  //return false if can't increment anymore
27  //TODO: DRY this code up with what's in ChainedAggregatorImpl
28  private boolean increment(List[] lengths, int[] indices, int j, int overrideIndex) {
29      if(j==-1) return false;
30      if(j==overrideIndex) {
31          return increment(lengths, indices, j-1, overrideIndex);
32      }
33      indices[j]++;
34      if(indices[j] >= lengths[j].size()) {
35          indices[j] = 0;
36          return increment(lengths, indices, j-1, overrideIndex);
37      }
38      return true;
39  }
```

- indices为要发送的消息在sides中的下标，算法中通过不断更新下标数组indices来完成叉积。
- 第3~6行初始化indices数据，使其指向sides中第0条消息。算法的基本思想是：固定overrideIndex对应的流，并且遍历其他流中已有的消息，overrideIndex所在的流的消息固定为overrideTuple，而其他流的消息则根据下标得到。

例如，3个流进行连接，S_1目前的消息为<A>, S_2为<>, S_3的消息为<C, D>。

(1) 若收到S_2的消息B，此时收到了3个流的消息，可以进行连接。

消息数组为：

S_1:<A>, S_2:<B>, S_3:<C, D>。

输出的消息为：

<A, B, C>, <A, B, D>。

(2) 若接下来收到S_2的消息E，消息数组为：

S_1:<A>, S_2:<B, E>, S_3:<C, D>。

则输出如下的消息：

<A, E, C>, <A, E, D>。

(3) 若接着收到S_1的消息F，消息数组为：

S_1:<A，F>, S_2:<B, E>, S_3:<C, D>。

输出的消息为：

<F, B, C>, <F, B, D>, <F, E, C>, <F, E, D>。

最终输出的消息为：

<A, B, C>, <A, B, D>,<A, E, C>, <A, E, D>,<F, B, C>, <F, B, D>, <F, E, C>, <F, E, D>。这与收到所有消息之后再进行叉积的效果是相同的，但实现了在每次收到新的消息时，就立

刻将满足连接条件的消息发送出去的效果。

22.2.3　complete方法

由于execute方法已经完成了内连接，complete方法则用来完成外连接操作，相关代码如下：

```
1 public void complete(JoinState state, TridentTuple group, TridentCollector collector) {
2     List<List>[] sides = state.sides;
3     boolean wasEmpty = state.numSidesReceived < sides.length;
4     for(int i=0; i<sides.length; i++) {
5         if(sides[i].isEmpty() && _types.get(i) == JoinType.OUTER) {
6             state.numSidesReceived++;
7             sides[i].add(makeNullList(_sideFields.get(i).size()));
8         }
9     }
10    if(wasEmpty && state.numSidesReceived == sides.length) {
11        emitCrossJoin(state, collector, -1, null);
12    }
13 }
14
15 @Override
16 public void cleanup() {
17 }
18
19 private List<Object> makeNullList(int size) {
20     List<Object> ret = new ArrayList(size);
21     for(int i=0; i<size; i++) {
22         ret.add(null);
23     }
24     return ret;
25 }
```

- 第3行的wasEmpty变量是重要的。若它为真，则表示从各个流都收到了消息并且完成了内连接，则complte方法不需要再做任何操作。若wasEmpty为假，则表明没有从某些源流收到消息，内连接没有完成。
- 第4~9行判断每一个源流的连接类型，若为OUTER类型，则调用makeNullList来补充一条空的消息。
- 第10~12行，若经过添加空消息，使得所有流都有了对应的消息，则调用emitCrossJoin来完成叉积并发送结果。注意，overrideIndex、overrideTuple分别传入了 - 1和空，表明将遍历所有流中的所有消息。

22.3　GroupedMultiReducerExecutor

JoinerMultiReducer假定收到的消息均是满足连接条件的，即分组域group列对应的值均相同，Trident通过GroupedMultiReducerExecutor来实现这一点。这个类的代码如下：

```
1 public class GroupedMultiReducerExecutor implements MultiReducer<Map<TridentTuple, Object>> {
2     GroupedMultiReducer _reducer;
3     List<Fields> _groupFields;
4     List<Fields> _inputFields;
5     List<ProjectionFactory> _groupFactories = new ArrayList<ProjectionFactory>();
6     List<ProjectionFactory> _inputFactories = new ArrayList<ProjectionFactory>();
7 
8     public GroupedMultiReducerExecutor(GroupedMultiReducer reducer, List<Fields> groupFields,
          List<Fields> inputFields) {
9         if(inputFields.size()!=groupFields.size()) {
10            throw new IllegalArgumentException("Multireducer groupFields and inputFields must
                  be the same size");
11        }
12        _groupFields = groupFields;
13        _inputFields = inputFields;
14        _reducer = reducer;
15    }
16 
17    @Override
18    public void prepare(Map conf, TridentMultiReducerContext context) {
19        for(int i=0; i<_groupFields.size(); i++) {
20            _groupFactories.add(context.makeProjectionFactory(i, _groupFields.get(i)));
21            _inputFactories.add(context.makeProjectionFactory(i, _inputFields.get(i)));
22        }
23        _reducer.prepare(conf, new TridentMultiReducerContext((List) _inputFactories));
24    }
25 
26    @Override
27    public Map<TridentTuple, Object> init(TridentCollector collector) {
28        return new HashMap();
29    }
30 
31    @Override
32    public void execute(Map<TridentTuple, Object> state, int streamIndex, TridentTuple full,
          TridentCollector collector) {
33        ProjectionFactory groupFactory = _groupFactories.get(streamIndex);
34        ProjectionFactory inputFactory = _inputFactories.get(streamIndex);
35 
36        TridentTuple group = groupFactory.create(full);
37        TridentTuple input = inputFactory.create(full);
38 
39        Object curr;
40        if(!state.containsKey(group)) {
41            curr = _reducer.init(collector, group);
42            state.put(group, curr);
43        } else {
44            curr = state.get(group);
45        }
46        _reducer.execute(curr, streamIndex, group, input, collector);
47    }
48 
49    @Override
50    public void complete(Map<TridentTuple, Object> state, TridentCollector collector) {
51        for(Map.Entry e: state.entrySet()) {
```

```
52              TridentTuple group = (TridentTuple) e.getKey();
53              Object val = e.getValue();
54              _reducer.complete(val, group, collector);
55          }
56      }
57 }
```

- 从第1行可以看出进行聚集的数据结构为Map<TridentTuple, Object>类型，其中键为通过分组列创建的消息，即连接的键，值为Object类型，在JoinerMultiReducer中为JoinState类型。其成员变量和构造函数较为简单，这里不再讨论。
- 第27~29行的init方法中，新建了一个哈希值用来存储每个分组。
- 第32~47行实现execute方法。第33~37行根据streamIndex获得用于创建输入消息和分组消息的工厂类，并创建出分组消息group和输入消息input。group为连接的键，input为其他列。
- 第40~43行获得连接分组列所对应的数据。若为第一次连接到某分组的消息，则调用_reducer.init(collector, group)进行创建。对于JoinerMultiReducer，则创建一个JoinState对象。
- 第46行调用_reducer的execute方法。JoinerMultiReducer的exeute方法则将进行内连接。
- 第50~56行对每一个连接键所在的消息组，调用_recuder的complete方法来完成外连接，当然这里也要首先根据JOIN的类型来判断是否补充空消息。

22.4 MultiReducerProcessor

MultiReducerProcessor用来执行MultiReducer。MultiReducerProcessor继承自TridentProcessor，它可以被放置在Trident的处理节点中执行。下面我们来看一下MultiReducer Processor类的实现代码：

```
1 public class MultiReducerProcessor implements TridentProcessor {
2     MultiReducer _reducer;
3     TridentContext _context;
4     Map<String, Integer> _streamToIndex;
5     List<Fields> _projectFields;
6     ProjectionFactory[] _projectionFactories;
7     FreshCollector _collector;
8
9     public MultiReducerProcessor(List<Fields> inputFields, MultiReducer reducer) {
10        _reducer = reducer;
11        _projectFields = inputFields;
12    }
13
14    @Override
15    public void prepare(Map conf, TopologyContext context, TridentContext tridentContext) {
16        List<Factory> parents = tridentContext.getParentTupleFactories();
17        _context = tridentContext;
18        _streamToIndex = new HashMap<String, Integer>();
19        List<String> parentStreams = tridentContext.getParentStreams();
20        for(int i=0; i<parentStreams.size(); i++) {
```

```
21            _streamToIndex.put(parentStreams.get(i), i);
22        }
23        _projectionFactories = new ProjectionFactory[_projectFields.size()];
24        for(int i=0; i<_projectFields.size(); i++) {
25          _projectionFactories[i] = new ProjectionFactory(parents.get(i), _projectFields.get(i));
26        }
27        _collector = new FreshCollector(tridentContext);
28        _reducer.prepare(conf, new TridentMultiReducerContext((List) Arrays.asList
              (_projectionFactories)));
29    }
30
31    @Override
32    public void cleanup() {
33        _reducer.cleanup();
34    }
35
36    @Override
37    public void startBatch(ProcessorContext processorContext) {
38        _collector.setContext(processorContext);
39        processorContext.state[_context.getStateIndex()] = _reducer.init(_collector);
40    }
41
42    @Override
43   public void execute(ProcessorContext processorContext, String streamId, TridentTuple tuple) {
44        _collector.setContext(processorContext);
45        int i = _streamToIndex.get(streamId);
46        _reducer.execute(processorContext.state[_context.getStateIndex()], i,
               _projectionFactories[i].create(tuple), _collector);
47    }
48
49    @Override
50    public void finishBatch(ProcessorContext processorContext) {
51        _collector.setContext(processorContext);
52        _reducer.complete(processorContext.state[_context.getStateIndex()], _collector);
53    }
54
55    @Override
56    public Factory getOutputFactory() {
57        return _collector.getOutputFactory();
58    }
59 }
```

理解这个类的重点在于看它是如何适配MultiReducer接口的，即何时去调用MultiReducer中的各个方法。下面简要介绍这个类中的成员变量和方法。

- _streamToIndex存储了从流名字到流编号的映射关系，编号从0开始。在MultiReducer的execute方法中需要传入streamIndex形参。
- _projectFields和_projectionFactories用来根据输入的消息创建连接要处理的消息。
- 第15~29行实现prepare方法。通过TridentContext获得该节点的父节点所对应的流，并对这些流执行连接操作。prepare方法中对这些流进行了编号，并调用_reducer的prepare方法。
- startBatch方法对该Processor的数据进行了初始化。这里调用了_reducer的init方法获得，

对于GroupedMultiReducerExecutory和JoinerMultiReducer实现来讲，init方法返回了HashMap<TridentTuple, JoinState>对象。

- 第43~47行的execute方法会根据输入消息的流号获得流的编号，并调用_reducer的execute方法。类似地，finishBatch方法中会调用_reducer的complete方法。
- MultiReducerProcessor的输入要保证具有相同连接分组列值的消息都到达相同的节点，这是通过分区节点来完成的。

22.5 连接操作

Join操作主要由下面4个函数来完成，它们的代码如下：

```
1 public Stream join(List<Stream> streams, List<Fields> joinFields, Fields outFields,
      List<JoinType> mixed) {
2     return multiReduce(strippedInputFields(streams, joinFields),
3           groupedStreams(streams, joinFields),
4           newJoinerMultiReducer(mixed, joinFields.get(0).size(),
              strippedInputFields(streams, joinFields)),
5           outFields);
6  }
7
8  private static List<GroupedStream> groupedStreams(List<Stream> streams,
       List<Fields> joinFields) {
9      List<GroupedStream> ret = new ArrayList<GroupedStream>();
10     for(int i=0; i<streams.size(); i++) {
11         ret.add(streams.get(i).groupBy(joinFields.get(i)));
12     }
13     return ret;
14 }
15
16 public Stream multiReduce(List<Fields> inputFields, List<GroupedStream> groupedStreams,
       GroupedMultiReducer function, Fields outputFields) {
17     List<Fields> fullInputFields = new ArrayList<Fields>();
18     List<Stream> streams = new ArrayList<Stream>();
19     List<Fields> fullGroupFields = new ArrayList<Fields>();
20     for(int i=0; i<groupedStreams.size(); i++) {
21         GroupedStream gs = groupedStreams.get(i);
22         Fields groupFields = gs.getGroupFields();
23         fullGroupFields.add(groupFields);
24         streams.add(gs.toStream().partitionBy(groupFields));
25         fullInputFields.add(TridentUtils.fieldsUnion(groupFields, inputFields.get(i)));
26
27     }
28     return multiReduce(fullInputFields, streams, new GroupedMultiReducerExecutor(function,
           fullGroupFields, inputFields), outputFields);
29 }
30
31 public Stream multiReduce(List<Fields> inputFields, List<Stream> streams, MultiReducer
       function, Fields outputFields) {
32     List<String> names = new ArrayList<String>();
33     for(Stream s: streams) {
```

```
34          if(s._name!=null) {
35              names.add(s._name);
36          }
37      }
38      Node n = new ProcessorNode(getUniqueStreamId(), Utils.join(names, "-"), outputFields,
             outputFields, new MultiReducerProcessor(inputFields, function));
39      return addSourcedNode(streams, n);
40  }
```

- 第8~14行，groupedStreams函数将输入的流转化为分组流。
- 第11行通过调用流的groupBy操作来完成向分组流的转化过程，分组域为进行连接操作的连接域。
- 第1~6行定义连接操作。strippedInputField函数用来将输入消息中进行连接的列去除。groupedStreams函数将输入的流转化为分组流。连接操作通过调用第16行的multiReduce操作来完成，实际进行连接的对象为JoinerMultiReducer。
- 第16~29行为multiReduce操作在分组流上的版本。第17~27行准备输入的流等，第24行调用GroupedStream的partitionBy操作创建新的流对象。此处将添加分区节点，具有相同groupFields值的消息将到达同一个节点上去。然后调用第31~40行定义的multiReduce方法，利用GroupedMultiReducerExecutor对JoinerMultiReducer进行封装。
- 第31~40行为multiReduce操作对应于基本流的版本，它含有一个处理节点，该节点使用MultiReducerProcessor来完成操作。
- 第39行，addSourcedNode函数将流添加到节点n的边，streams为所有要进行操作的流。

为了便于理解，现将连接操作的实现过程绘制成图，如图22-2所示，并给出步骤归纳。

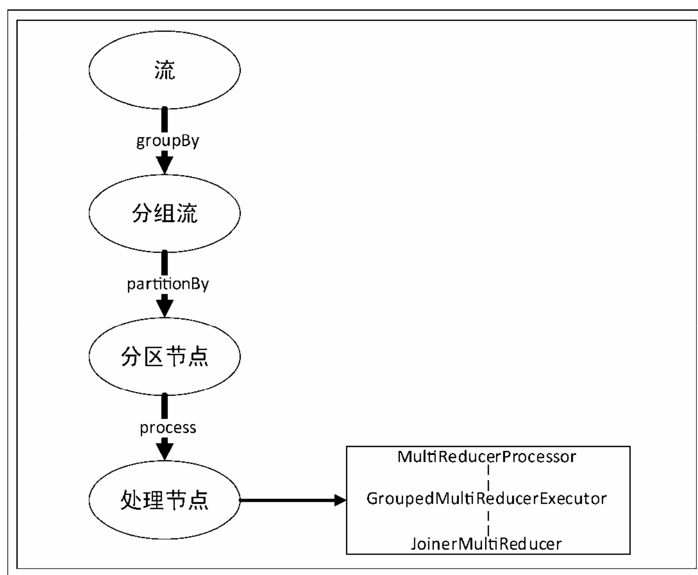


图22-2　连接操作的实现

- 要进行连接的流通过groupBy变为分组流；
- 通过调用partitionBy添加分区节点；
- 在分区节点之后添加处理节点，该处理节点完成对流的连接操作。

22.6 流合并操作

merge操作将输入的各个流的消息合并在一起。各个流的消息格式需要保持统一。merge操作不需要分区节点的协助。

merge操作同样需调用multiReduce来完成实现，它采用了IdentityMultiReducer对象。merge操作的代码如下：

```
public Stream merge(Fields outputFields, List<Stream> streams) {
    return multiReduce(streams, new IdentityMultiReducer(), outputFields);
}
```

IdentityMultiReducer的实现是简单的，它只是将收到的消息发送了出去，其代码如下：

```
public class IdentityMultiReducer implements MultiReducer {
    @Override
    public void prepare(Map conf, TridentMultiReducerContext context) {
    }
    @Override
    public Object init(TridentCollector collector) {
        return null;
    }
    @Override
    public void execute(Object state, int streamIndex, TridentTuple input, TridentCollector collector) {
        collector.emit(input);
    }
    @Override
    public void complete(Object state, TridentCollector collector) {
    }
    @Override
    public void cleanup() {
    }
}
```

第 23 章

Trident中的Bolt节点

SubTopologyBolt类型为Trident中运行的基本单位，但它并不是真正的Bolt节点，Trident会利用TridentBoltExecutor对SubTopologyBolt进行接口适配。TridentBoltExecutor继承自IRichBolt接口，是Trident中真正运行的Bolt节点。它提供了类似于协调Bolt（CoordinatedBolt）节点的功能，通过发送协调消息来对各个节点进行同步。

SubTopologyBolt主要用于对TridentProcessor的执行进行抽象。本章将讨论SubTopologyBolt和TridentBoltExecutor的实现。

23.1 SubTopologyBolt

Trident可以把一些运算放在同一个Bolt中来执行，这个Bolt的类型就是SubTopologyBolt。它是Trident中的基本执行单位。本节对SubTopologyBolt进行详细分析。

23.1.1 输入准备

InitialReceiver类用于为SubTopologyBolt准备输入，其实现代码如下：

```
1  protected class InitialReceiver {
2      List<TridentProcessor> _receivers = new ArrayList();
3      RootFactory _factory;
4      ProjectionFactory _project;
5      String _stream;
6
7      public InitialReceiver(String stream, Fields allFields) {
8          // TODO: don't want to project for non-batch bolts...???
9          // how to distinguish "batch" streams from non-batch streams?
10         _stream = stream;
11         _factory = new RootFactory(allFields);
12         List<String> projected = new ArrayList(allFields.toList());
13         projected.remove(0);
14         _project = new ProjectionFactory(_factory, new Fields(projected));
15     }
16
17     public void receive(ProcessorContext context, Tuple tuple) {
18     TridentTuple t = _project.create(_factory.create(tuple));
19         for(TridentProcessor r: _receivers) {
```

```
20            r.execute(context, _stream, t);
21        }
22    }
23
24    public void addReceiver(TridentProcessor p) {
25        _receivers.add(p);
26    }
27
28    public Factory getOutputFactory() {
29        return _project;
30    }
31 }
```

- _factory：RootFactory对象，用于将输入的Tuple类型的消息适配为TridentTuple类型的消息。
- _project：ProjectionFactory对象，根据输入的需要对字段名进行映射，它的输入即是RootFactory的输出。
- _receivers：为TridentProcessor的集合，它将对_project产生的消息依次调用execute方法。
- _stream：表示输入的消息的StreamId。
- 第17~22行的receive方法将对输入的消息先调用_factory和_project的create方法，然后调用TridentProcessor的execute方法。
- 第13行去除输入的消息的第1列，该列为事务序号。由于该列所对应的值会反映在ProcessorContext中，故这里不再进行处理。

23.1.2 成员变量

类SubTopologyBolt的定义如下：

```
public class SubtopologyBolt implements ITridentBatchBolt {
    DirectedGraph _graph;
    Set<Node> _nodes;
    Map<String, InitialReceiver> _roots = new HashMap();
    Map<Node, Factory> _outputFactories = new HashMap();
    Map<String, List<TridentProcessor>> _myTopologicallyOrdered = new HashMap();
    Map<Node, String> _batchGroups;
}
```

首先来看类中的各个成员变量。

- _graph：整个Topology所对应的有向图。
- _nodes：该Bolt中所包含的处理节点。_nodes为_graph节点的子集。
- _roots：每种类型的输入流都会对应一个InitalReceiver对象，用于表示如何处理该流的消息。_roots对应于Subtopology的外部输入。
- _outputFactories：每个处理节点都会对应一个输出的工厂。
- _myTopologicallyOrdered：它的键为节点组序号，值为节点组所对应的TridentProcessor。这些处理节点的顺序为拓扑排序的结果，这是很关键的。

23

- _batchGroups：该变量保存了反向的索引，用来表示每个节点属于哪一个节点组。
- BatchGroup对应于_graph中的一个最大连通子图。例如，在含有DRPC的Topology中，DRPC的节点与其他节点是不连通的，它们将属于不同的节点组。
- 在一个SubTopologyBolt中，含有多个节点组是可能的。例如在含有DRPC的Topology中，查询操作与存储操作可以被分配到同一个SubTopologyBolt中，于是该Bolt可能收到来自两个节点组的消息。

23.1.3　主要方法

prepare方法是SubTopologyBolt类中最为核心的方法，其代码如下：

```
1 public void prepare(Map conf, TopologyContext context, BatchOutputCollector batchCollector) {
2     int thisComponentNumTasks = context.getComponentTasks(context.getThisComponentId()).size();
3     for(Node n: _nodes) {
4         if(n.stateInfo!=null) {
5             State s = n.stateInfo.spec.stateFactory.makeState(conf, context,
                  context.getThisTaskIndex(), thisComponentNumTasks);
6             context.setTaskData(n.stateInfo.id, s);
7         }
8    }
9    DirectedSubgraph<Node, Object> subgraph = new DirectedSubgraph(_graph, _nodes, null);
10   TopologicalOrderIterator it = new TopologicalOrderIterator<Node, Object>(subgraph);
11   int stateIndex = 0;
12   while(it.hasNext()) {
13       Node n = (Node) it.next();
14       if(n instanceof ProcessorNode) {
15           ProcessorNode pn = (ProcessorNode) n;
16           String batchGroup = _batchGroups.get(n);
17           if(!_myTopologicallyOrdered.containsKey(batchGroup)) {
18               _myTopologicallyOrdered.put(batchGroup, new ArrayList());
19           }
20           _myTopologicallyOrdered.get(batchGroup).add(pn.processor);
21           List<String> parentStreams = new ArrayList();
22           List<Factory> parentFactories = new ArrayList();
23           for(Node p: TridentUtils.getParents(_graph, n)) {
24               parentStreams.add(p.streamId);
25               if(_nodes.contains(p)) {
26                   parentFactories.add(_outputFactories.get(p));
27              } else {
28                   if(!_roots.containsKey(p.streamId)) {
29                       _roots.put(p.streamId, new InitialReceiver(p.streamId,
                              getSourceOutputFields(context, p.streamId)));
30                   }
31                   _roots.get(p.streamId).addReceiver(pn.processor);
32                   parentFactories.add(_roots.get(p.streamId).getOutputFactory());
33               }
34           }
35           List<TupleReceiver> targets = new ArrayList();
36           boolean outgoingNode = false;
37           for(Node cn: TridentUtils.getChildren(_graph, n)) {
```

```
38              if(_nodes.contains(cn)) {
39                  targets.add(((ProcessorNode) cn).processor);
40              } else {
41                  outgoingNode = true;
42              }
43          }
44          if(outgoingNode) {
45              targets.add(new BridgeReceiver(batchCollector));
46          }
47
48          TridentContext triContext = new TridentContext(
49                  pn.selfOutFields,
50                  parentFactories,
51                  parentStreams,
52                  targets,
53                  pn.streamId,
54                  stateIndex,
55                  batchCollector
56                  );
57          pn.processor.prepare(conf, context, triContext);
58          _outputFactories.put(n, pn.processor.getOutputFactory());
59      }
60      stateIndex++;
61  }
62  // TODO: get prepared one time into executor data... need to avoid the ser/deser
63  // for each task (probably need storm to support boltfactory)
64 }
```

- 第2~8行判断节点的stateInfo是否为空，并对存储的State对象进行初始化。State类是Trident对存储的一层抽象，也是Trident的核心部分，后面的章节将对其进行更为详细的讨论。初始化过后的State对象被存储于TopologyContext的taskData中，并以stateInfo.id为键。stateInfo.id是一个以串“state”为前缀的在Topology中唯一的字符串。
- 第9行根据SubTopologyBolt中包含的节点获得一个子图。
- 第10行对子图进行拓扑排序，TopologicalOrderIterator类型的it变量用来按照拓扑排序的顺序遍历子图。
- 第14行，可以看到SubTopologyBolt只对处理节点进行操作。处理节点中包含了一个TridentProcessor。Spout节点和分区节点则不在SubTopologyBolt的处理范围之内。
- 第15~19行依照节点组序号的顺序，将节点中所含有的处理器放入_myTopologicallyOrdered变量中。例如，当SubTopologyBolt要对一个事务进行处理时，它将按照_myTopologicallyOrdered中的顺序依次调用TridentProcessor的initBatch和finishBatch方法。
- 第20~34行处理TridentProcessor的输入。parentStreams为TridentProcessor输入流。parentFactories为这些流所对应的工厂。第23行获得该节点在图中对应的所有父节点。第25~26行表示该父节点为SubTopologyBolt对应的子图中的某个节点，由于拓扑排序，这个节点已经在之前处理过了，故可以直接得到该流所对应的输出工厂。第28~30行表示该节点为外部节点，于是根据其流序号以及输出的字段名构建一个InitialReceiver对象，并将其

放在_roots中，表示该流来自于外部节点。第31行将当前节点所对应的TridentProcessor作为该流的一个接收端。

- 第35~43行处理该节点中TridentProcessor的输出。第35行定义的targets变量记录了“哪些节点的TridentProcessor将处理该节点的输出”这一信息。如果输出的节点不在SubTopologyBolt所对应的子图中，则将BridgeReceiver作为其目标节点。
- 第48~56行构建了TridentContext对象，可以看出TridentContext对象是与TridentProcessor一一对应的，该对象包含了TridentProcessor所需要的上下文环境（例如，处理器的输入、输出，将要处理的流等等）。
- 第57行调用该TridentProcessor的prepare方法，它将新产生的TridentContext对象作为构造函数的一个参数传入。
- 第58行将该TridentProcessor所对应的输出添加到SubTopologyBolt的输出中。于是该输出便可被其他的SubTopologyBolt作为输入了。
- stateIndex变量用于唯一地标识SubTopologyBolt中的每一个节点。
- 第62~63行提出了一些建议，希望将该方法中较为通用的部分放置于Executor的共享数据中，这样每一个Task就不需要再对它们进行重复计算。

接下来看execute方法的实现：

```java
@Override
public void execute(BatchInfo batchInfo, Tuple tuple) {
    String sourceStream = tuple.getSourceStreamId();
    InitialReceiver ir = _roots.get(sourceStream);
    if(ir==null) {
        throw new RuntimeException("Received unexpected tuple " + tuple.toString());
    }
    ir.receive((ProcessorContext) batchInfo.state, tuple);
}
```

- 首先，根据输入消息的流号，在_roots中找到对应的InitialReceiver对象，并调用其receive方法。
- 根据前面的讨论可以知道，所有等待该流消息的TridentProcessor的execute方法均会被调用。
- 在某个TridentProcessor的execute方法中，下游TridentProcessor的execute方法也会被依次调用到，于是构成了一个调用链，直到SubTopologyBolt完成对该消息的处理后结束。

finishBatch和initBatchState方法会被依照拓扑排序的节点顺序依次调用：

```java
@Override
public void finishBatch(BatchInfo batchInfo) {
    for(TridentProcessor p: _myTopologicallyOrdered.get(batchInfo.batchGroup)) {
        p.finishBatch((ProcessorContext) batchInfo.state);
    }
}

@Override
public Object initBatchState(String batchGroup, Object batchId) {
```

```
        ProcessorContext ret = new ProcessorContext(batchId, new Object[_nodes.size()]);
        for(TridentProcessor p: _myTopologicallyOrdered.get(batchGroup)) {
            p.startBatch(ret);
        }
        return ret;
    }
```

在initBatchState方法中，会对ProcessorContext的数据进行初始化，然后返回ProcessorContext对象。Trident中的聚集器均要基于ProcessorContext中state所存储的数据来实现。

declareOutputFields方法会对SubTopologyBolt中每一个节点的输出进行声明，它将$batchId作为第1列。可以看出SubTopologyBolt虽然作为一个整体而存在，可其中每一个节点的输出均可能成为最终的输出。目前，只有BrigeReceiver的节点才会真正发送消息。declareOutput Fields方法的代码如下：

```
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        for(Node n: _nodes) {
            declarer.declareStream(n.streamId, TridentUtils.fieldsConcat(new Fields("$batchId"),
                n.allOutputFields));
        }
    }
```

类BridgeReceiver的定义如下：

```
    public class BridgeReceiver implements TupleReceiver {

        BatchOutputCollector _collector;

        public BridgeReceiver(BatchOutputCollector collector) {
            _collector = collector;
        }

        @Override
        public void execute(ProcessorContext context, String streamId, TridentTuple tuple) {
            _collector.emit(streamId, new ConsList(context.batchId, tuple));
        }
    }
```

BridgeReceiver实现了TupleReceiver接口，其execute方法负责将输入的消息发送出去，它将ProcessorContext中的batchId作为第1列。

其他方法都比较直观，这里不再一一讨论。

23.2 Trident 中的 Bolt 执行器

类似于事务Topology中的协调Bolt，Trident中利用TridentBoltExecutor来执行Trident中的SubTopologyBolt。

TridentBoltExecutor的实现与CoordinatedBolt的实现非常类似，不过它设计得更加灵活健

壮。读者可以参考CooridnatedBolt的实现来分析TridentTopologyBolt，并将二者进行对比，由此可观察到Storm的设计演化。

23.2.1　ITridentBatchBolt接口

接口ITridentBatchBolt的定义如下：

```
public interface ITridentBatchBolt extends IComponent {
    void prepare(Map conf, TopologyContext context, BatchOutputCollector collector);
    void execute(BatchInfo batchInfo, Tuple tuple);
    void finishBatch(BatchInfo batchInfo);
    Object initBatchState(String batchGroup, Object batchId);
    void cleanup();
}
```

与IBatchBolt接口相比，这个接口多了initBatchState方法和cleanup方法。在BatchBolt Executor中，为了更好地隔离各个事务，每次都需要反序列化产生一个新的Bolt对象。Trident避免了这种情况的发生，在收到节点组中一个事务的第一条消息后，将调用initBatchState方法初始化事务，并在结束时调用finishBatch方法，从而实现了事务隔离。

23.2.2　TrackedBatch

SubTopologyBolt节点中可能会同时处理来自多个事务的消息，类TrackedBatch用于跟踪Bolt中正在处理的事务。该类的数据成员分析如下：

```
public static class TrackedBatch {
    int attemptId;
    BatchInfo info;
    CoordCondition condition;
    int reportedTasks = 0;
    int expectedTupleCount = 0;
    int receivedTuples = 0;
    Map<Integer, Integer> taskEmittedTuples = new HashMap();
    boolean failed = false;
    boolean receivedCommit;
    Tuple delayedAck = null;

    public TrackedBatch(BatchInfo info, CoordCondition condition, int attemptId) {
        this.info = info;
        this.condition = condition;
        this.attemptId = attemptId;
        receivedCommit = condition.commitStream == null;
    }
}
```

- attemptId为事务尝试序号，在Trident中为一个递增的值。
- info为BatchInfo对象。BatchInfo的定义如下：

```
public class BatchInfo {
    public IBatchID batchId;
    public Object state;
    public String batchGroup;
}
```

- batchId目前与事务尝试序号是等同的。在Trident中，处理的数据都是分批的，可以将其理解成为事务。DRPC Spout发送的消息为查询消息，这与事务的概念就并不一致了。但事实上，也可以将DRPC Spout发送的消息及其衍生消息认为是事务，只不过它们不具备重传的功能。因此本书对这两种类型不加区分，统称其为事务。
- state为这个事务所对应的数据，在initBatchState时获得。
- batchGroup为该事务所在的节点组。

下面回到类TrackedBatch。

- condition为CoordCondition对象，其定义如下：

```
public static class CoordCondition implements Serializable {
    public GlobalStreamId commitStream;
    public int expectedTaskReports;
    Set<Integer> targetTasks;
}
```

- commitStream为一个全局流对象，代表这个节点组中对应的事务提交流。
- expectedTaskReports表示直接上游节点的个数，本节点将从直接上游节点接收数据。从每一个节点都收到协调消息是事务在该节点上处理结束的条件之一。
- targetTasks表示直接下游节点列表，本节点将向直接下游节点发送协调信息。

CoordCondition的抽象更加直观：一个事务在该节点是否已经完成，取决于三方面条件，首先是所有的上游节点都已经向该节点发送了协调信息，其次是收到了从事务提交流发送过来的事务提交消息，最后是该节点上收到了确定的消息数目（该数目由协调消息给出）。

再回到TrackedBatch。

- reportedTasks表示已经向该节点发送过协调消息的节点数目。
- expectedTupleCount表示通过协调信息计算得到的，也即为该节点需要收到的消息数目。
- receivedTuples表示已经收到的消息条数。它将与expectedTupleCount变量进行比较，来判断事务是否满足结束条件。
- taskEmittedTuples表示已经向下游节点发送的消息条数。
- receivedCommit表示是否已经从事务提交流收到了事务提交消息。
- delayedAck是比较有趣的：为了保证消息的可靠传输，它将保留至少一个协调消息，在所有的消息均已经发送出去后，再对保留的消息进行Ack。
- failed用于表明当前事务是否已经处理失败。

通常情况下，一个上游节点通过直接消息流来向每一个下游节点发送协调消息，即便上游节点未向其中任何一个下游节点发送任何一条消息（即协调消息中的消息数目域为0），也必须发送

该协调消息。例如，系统中存在节点类型A和节点类型B，A的并行度为10，B的并行度为5，则B的每一个节点都需要收到A中每一个节点的协调消息才有可能认为处理结束，即B的每一个节点都需要收到10条协调消息。

但是在某些情况下，上游节点若确定为一个事务只由其中的某一个节点产生时，下游节点则只需要收到一条协调消息即可。例如， DRPC Spout A的并行度为10，类型B的并行度为5。我们知道，DRPC Spout发送的每条消息都是独立请求的，于是B中每一个处理节点只要收到从任意一个DRPC Spout发送的协调消息即可，即B的每一个节点只需要收到1条协调消息。

类CoordSpec用来描述当前节点与上游节点的关系，其定义如下：

```
public static class CoordSpec implements Serializable {
    public GlobalStreamId commitStream = null;
    public Map<String, CoordType> coords = new HashMap<String, CoordType>();

    public CoordSpec() {
    }
}
```

- commitStream与CoordCondition中的情况一致，表示为全局事务提交流。
- coords用来描述每一个上游节点需要的协调消息数目。
- CoordType用来描述协调消息的类型，由直接上游节点的个数决定。若仅有一个上游节点，则只需要收到一条协调消息即可；若含有多个上游节点，则需要收到所有上游节点的协调消息。

23.2.3　定制的输出收集器

TridentBoltExecutor中需要对发送至每个目标节点的消息数目进行统计。Trident对基础的输出器进行了包装，使其可以在发送消息的同时，更新发送消息的数目。该数目最终将通过协调消息发送至下游目标节点。类CoordinatedOutputCollector的代码如下：

```
1 public class CoordinatedOutputCollector implements IOutputCollector {
2     IOutputCollector _delegate;
3
4     TrackedBatch _currBatch = null;;
5
6     public void setCurrBatch(TrackedBatch batch) {
7         _currBatch = batch;
8     }
9
10    public CoordinatedOutputCollector(IOutputCollector delegate) {
11        _delegate = delegate;
12    }
13
14    public List<Integer> emit(String stream, Collection<Tuple> anchors, List<Object> tuple) {
15        List<Integer> tasks = _delegate.emit(stream, anchors, tuple);
16        updateTaskCounts(tasks);
```

```
17        return tasks;
18    }
19
20    public void emitDirect(int task, String stream, Collection<Tuple> anchors,
          List<Object> tuple) {
21        updateTaskCounts(Arrays.asList(task));
22        _delegate.emitDirect(task, stream, anchors, tuple);
23    }
24
25    public void ack(Tuple tuple) {
26        throw new IllegalStateException("Method should never be called");
27    }
28
29    public void fail(Tuple tuple) {
30        throw new IllegalStateException("Method should never be called");
31    }
32
33    public void reportError(Throwable error) {
34        _delegate.reportError(error);
35    }
36
37
38    private void updateTaskCounts(List<Integer> tasks) {
39        if(_currBatch!=null) {
40            Map<Integer, Integer> taskEmittedTuples = _currBatch.taskEmittedTuples;
41            for(Integer task: tasks) {
42                int newCount = Utils.get(taskEmittedTuples, task, 0) + 1;
43                taskEmittedTuples.put(task, newCount);
44            }
45        }
46    }
47 }
```


- 类CoordinatedOutputCollector中包含了一个TrackedBatch类型的_currBatch对象引用。
- 第38~46行的updateTaskCounts方法根据输入的目标Task列表，更新其发送至该目标节点的消息数目。
- 第25~31行的ack和fail实现方法为空，表明该输出收集器的主要目的是发送消息以及更新发送消息的数目，统计输入消息的数目不在其职责范围内，这与CoordinatedBolt的实现是不同的。
- 第6~8行，setCurrBatch方法将_currBatch绑定到当前事务，在收到一条消息时对它进行调用，并在处理该消息之后将其设为空，于是CoordinatedOutputCollector便可以同时对多个事务进行统计了。

最后，利用BatchOutputCollectorImpl和OutputCollector类对Collector进行包装。_coorOutputCollector内含更加灵活的重载方法并且具备统计功能，相关代码如下。

```
_coordCollector = new CoordinatedOutputCollector(collector);
_coordOutputCollector = new BatchOutputCollectorImpl(new OutputCollector(_coordCollector));
```

23.2.4　消息类型

根据消息的来源，Trident将消息划分为以下3种类型：

- REGULAR表示为通常的数据消息；
- COMMIT表示为事务提交消息；
- COORD表示为从上游节点发送过来的协调消息，内容为上游节点发送给当前节点的消息条数。

getTupleType用于计算输入消息的类型，相关的分析如下：

```
1 static enum TupleType {
2     REGULAR,
3     COMMIT,
4     COORD
5 }
6
7 private TupleType getTupleType(Tuple tuple, TrackedBatch batch) {
8     CoordCondition cond = batch.condition;
9     if(cond.commitStream!=null
10            && tuple.getSourceGlobalStreamid().equals(cond.commitStream)) {
11        return TupleType.COMMIT;
12    } else if(cond.expectedTaskReports > 0
13            && tuple.getSourceStreamId().startsWith(COORD_STREAM_PREFIX)) {
14        return TupleType.COORD;
15    } else {
16        return TupleType.REGULAR;
17    }
18 }
19
20 public static String COORD_STREAM_PREFIX = "$coord-";
21
22 public static String COORD_STREAM(String batch) {
23    return COORD_STREAM_PREFIX + batch;
24 }
```

- 所有协调消息的流都是以“$coord-”开头的，完整的协调流名字会在“$coord-”后面跟上具体的节点组序号，如$coord-bg0。
- 输入的消息来自$commit，消息为事务提交消息。
- 若期待上游节点数目（expectedTaskReports）大于零，且消息源的流以“$coord-”为前缀，则该消息为协调消息。如果收到了以“$coord-”为前缀的消息，但期待上游节点数目为零，我们认为应该抛出异常。
- 其他情况为正常的数据消息。

23.2.5　数据成员分析

本节对TridentBoltExecutor的数据成员以及prepare方法进行分析，相关代码如下：

```
1 Map<GlobalStreamId, String> _batchGroupIds;
2 Map<String, CoordSpec> _coordSpecs;
```

```
3 Map<String, CoordCondition> _coordConditions;
4 ITridentBatchBolt _bolt;
5 long _messageTimeoutMs;
6 long _lastRotate;
7 OutputCollector _collector;
8 CoordinatedOutputCollector _coordCollector;
9 BatchOutputCollector _coordOutputCollector;
10TopologyContext _context;
11
12 @Override
13 public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
14     _messageTimeoutMs = context.maxTopologyMessageTimeout() * 1000L;
15     _lastRotate = System.currentTimeMillis();
16     _batches = new RotatingMap(2);
17     _context = context;
18     _collector = collector;
19     _coordCollector = new CoordinatedOutputCollector(collector);
20    _coordOutputCollector = new BatchOutputCollectorImpl(new OutputCollector(_coordCollector));
21
22     _coordConditions = (Map) context.getExecutorData("__coordConditions");
23     if(_coordConditions==null) {
24         _coordConditions = new HashMap();
25         for(String batchGroup: _coordSpecs.keySet()) {
26             CoordSpec spec = _coordSpecs.get(batchGroup);
27             CoordCondition cond = new CoordCondition();
28             cond.commitStream = spec.commitStream;
29             cond.expectedTaskReports = 0;
30             for(String comp: spec.coords.keySet()) {
31                 CoordType ct = spec.coords.get(comp);
32                 if(ct.equals(CoordType.single())) {
33                     cond.expectedTaskReports+=1;
34                 } else {
35                     cond.expectedTaskReports+=context.getComponentTasks(comp).size();
36                 }
37             }
38             cond.targetTasks = new HashSet<Integer>();
39             for(String component: Utils.get(context.getThisTargets(),
40                                            COORD_STREAM(batchGroup),
41                                            new HashMap<String, Grouping>()).keySet()) {
42                 cond.targetTasks.addAll(context.getComponentTasks(component));
43             }
44             _coordConditions.put(batchGroup, cond);
45         }
46         context.setExecutorData("_coordConditions", _coordConditions);
47     }
48     _bolt.prepare(conf, context, _coordOutputCollector);
49 }
```

- _batchGroupsIds用来存储从全局流到节点组序号的映射关系。
- _coordSpec用来存储每个节点组内部节点协调消息的接收关系。
- _coordConditions用来存储每个节点组内部的协调条件。参考前面关于TrackedBatch的实现分析，可正确理解这两个成员变量中所含有的信息。

- _batches用来跟踪该节点正在处理的事务。
- _bolt为TridentBoltExecutor所代理的Bolt节点，目前为SubTopologyBolt类型。
- 第5~6行用来设置消息的超时，与系统的消息超时设置相同。
- 第7~9行用来设置系统的输出收集器，主要用于向外发送数据并进行统计。
- 成员方法prepare主要用来完成对成员变量的初始化。处于同一个Executor的Task会属于相同的组件，这些Task的协调消息设置是相同的。于是在第46行，Trident将该设置存储于Executor的共享数据中。由于Executor中的Task是按照顺序进行初始化的，故该Executor中的其他Task将直接获得这些设置，并不需要重新计算。
- 第29~37行根据CoordType来设置协调条件。Single类型的CoorType主要用于上游节点为Spout类型，且该Spout的每个事务只含有一条消息的情况，例如DRPC Spout。
- 第38~42行设置该节点的目标接收节点。

23.2.6　主要成员方法分析

成员方法execute是TridentBoltExecutor的核心方法，其代码如下：

```
1 public void execute(Tuple tuple) {
2     if(tuple.getSourceStreamId().equals(Constants.SYSTEM_TICK_STREAM_ID)) {
3         long now = System.currentTimeMillis();
4         if(now - _lastRotate > _messageTimeoutMs) {
5             _batches.rotate();
6             _lastRotate = now;
7         }
8         return;
9     }
10    String batchGroup = _batchGroupIds.get(tuple.getSourceGlobalStreamid());
11    if(batchGroup==null) {
12        // this is so we can do things like have simple DRPC that doesn't need to use
              batch processing
13        _coordCollector.setCurrBatch(null);
14        _bolt.execute(null, tuple);
15        _collector.ack(tuple);
16        return;
17    }
18    IBatchID id = (IBatchID) tuple.getValue(0);
19    //get transaction id
20    //if it already exissts and attempt id is greater than the attempt there
21
22
23    TrackedBatch tracked = (TrackedBatch) _batches.get(id.getId());
24
25    // this code here ensures that only one attempt is ever tracked for a batch, so when
26    // failures happen you don't get an explosion in memory usage in the tasks
27    if(tracked!=null) {
28        if(id.getAttemptId() > tracked.attemptId) {
29            _batches.remove(id.getId());
30            tracked = null;
```

```
31        } else if(id.getAttemptId() < tracked.attemptId) {
32            // no reason to try to execute a previous attempt than we've already seen
33            return;
34        }
35    }
36
37   if(tracked==null) {
38        tracked = new TrackedBatch(new BatchInfo(batchGroup, id,
              _bolt.initBatchState(batchGroup, id)), _coordConditions.get(batchGroup),
                  id.getAttemptId());
39        _batches.put(id.getId(), tracked);
40    }
41    _coordCollector.setCurrBatch(tracked);
42
43    TupleType t = getTupleType(tuple, tracked);
44    if(t==TupleType.COMMIT) {
45        tracked.receivedCommit = true;
46        checkFinish(tracked, tuple, t);
47    } else if(t==TupleType.COORD) {
48        int count = tuple.getInteger(1);
49        tracked.reportedTasks++;
50        tracked.expectedTupleCount+=count;
51        checkFinish(tracked, tuple, t);
52    } else {
53        tracked.receivedTuples++;
54        boolean success = true;
55        try {
56            _bolt.execute(tracked.info, tuple);
57            if(tracked.condition.expectedTaskReports==0) {
58                success = finishBatch(tracked, tuple);
59            }
60        } catch(FailedException e) {
61            failBatch(tracked, e);
62        }
63        if(success) {
64            _collector.ack(tuple);
65        } else {
66            _collector.fail(tuple);
67        }
68       }
69       _coordCollector.setCurrBatch(null);
70   }
```

- Bolt需要跟踪所有正在被处理的事务，但由于某些事务处理失败，被跟踪的事务可能不会被及时清理，长此下去将导致内存泄露。代码的第2~9行对这种情况进行了处理，这非常类似于Spout节点中的消息超时技术，即当收到从Tick流发来的消息时，系统会对_batch中较老的数据进行超时处理。
- 第10~17行，若消息的来源流不属于任何一个节点组，则不对该消息进行跟踪，而是直接调用代理类的execute方法，并对输入的消息进行Ack。
- 第18~40行，若为该事务的第一条消息或者该消息所在事务的尝试序号更大，则创建一个

新的事务跟踪对象tracked。若消息所在的事务尝试编号比目前正在跟踪的事务的尝试编号小，则表明无需对该消息进行处理。第38行调用Bolt的initBatchState方法对事务所对应的数据进行初始化，具体地，就是调用每个处理节点的startBatch方法。

- 当收到控制消息时，将检查事务在该节点处是否已经处理结束。第44~46行，若收到的消息类型为事务提交消息，则设置receivedCommit为true，这是事务处理结束的条件之一。第47~51行，若收到的消息为协调消息类型，则更新收到的协调消息数目和收到的消息总数。
- 第53~67行对普通的数据消息进行处理并完成Ack。注意，若期待消息来源数目为零，则直接调用finishBatch方法。
- 由于一个节点可能同时处理来自多个事务的消息，故第41行在处理消息前设置上下文对象，第 69行在处理完成后设置上下文为空。这是进行消息发送数目统计的关键。

在收到控制消息后，将调用checkFinish方法来检查事务处理是否已经结束，这个方法的代码如下：

```
1 private void checkFinish(TrackedBatch tracked, Tuple tuple, TupleType type) {
2     if(tracked.failed) {
3         failBatch(tracked);
4         _collector.fail(tuple);
5         return;
6     }
7 
8     CoordCondition cond = tracked.condition;
9     boolean delayed = tracked.delayedAck==null &&
10        (cond.commitStream!=null && type==TupleType.COMMIT
11        || cond.commitStream==null);
12    if(delayed) {
13        tracked.delayedAck = tuple;
14    }
15    boolean failed = false;
16    if(tracked.receivedCommit && tracked.reportedTasks == cond.expectedTaskReports) {
17        if(tracked.receivedTuples == tracked.expectedTupleCount) {
18            //Take the tuple as the Anchor
19            finishBatch(tracked, tuple);
20        } else {
21            //TODO: add logging that not all tuples were received
22            failBatch(tracked);
23            _collector.fail(tuple);
24            failed = true;
25        }
26    }
27 
28    if(!delayed && !failed) {
29        _collector.ack(tuple);
30    }
31 }
```

- 第8~14行，若该节点为事务提交节点，则将从事务提交流收到的消息作为最后被Ack的控制消息。否则，就选取第一个到达的协调消息作为最后被Ack的控制消息。只有当属于一

个事务的消息全都被处理完成并发送后，才会对这条控制消息进行Ack，这是保证Ack框架正常工作所必须的步骤。

- 第16~26行，若已经收到的协调消息数目与期望相同，期望收到的消息数目与实际收到的消息数目相同（若为事务提交节点，则还需收到事务提交消息），那么此时该节点对事务的处理可以结束。接下来，调用finishBatch方法对事务进行后处理。协调消息其实是最后发送的，若已经收到了所有的协调消息，但仍有消息未收到，则直接对事务进行失败处理。

下面我们来看finishBatch的方法实现，其逻辑是直观的，代码如下：

```
1 private boolean finishBatch(TrackedBatch tracked, Tuple finishTuple) {
2     boolean success = true;
3     try {
4         _bolt.finishBatch(tracked.info);
5         String stream = COORD_STREAM(tracked.info.batchGroup);
6         for(Integer task: tracked.condition.targetTasks) {
7             _collector.emitDirect(task, stream, finishTuple, new Values(tracked.info.batchId,
                  Utils.get(tracked.taskEmittedTuples, task, 0)));
8         }
9         if(tracked.delayedAck!=null) {
10            _collector.ack(tracked.delayedAck);
11            tracked.delayedAck = null;
12        }
13    } catch(FailedException e) {
14        failBatch(tracked, e);
15        success = false;
16    }
17    _batches.remove(tracked.info.batchId.getId());
18    return success;
19 }
```

- 第4行调用代理Bolt的finishBatch方法。
- 第5~9行向下游节点发送协调消息。
- 第9~12行对最后的控制消息进行Ack。
- 第17行，去掉对该事务的跟踪。

failBatch方法较为直观，代码如下：

```
private void failBatch(TrackedBatch tracked, FailedException e) {
    if(e!=null && e instanceof ReportedFailedException) {
        _collector.reportError(e);
    }
    tracked.failed = true;
    if(tracked.delayedAck!=null) {
        _collector.fail(tracked.delayedAck);
        tracked.delayedAck = null;
    }
}
```

23

第 24 章

Trident的执行优化

Trident中采用了较多图的算法，来对执行的Topology结构进行优化，与Bolt或Spout作为最小运算单元的传统Topology不同，Trident中定义了更细微的执行单位。Trident的执行优化会尽可能多地将运算放置在一个SubTopologyBolt或Spout节点中执行，以提高运算效率，降低网络传输消耗。

本章将首先讨论Trident中的节点类型，然后介绍执行优化算法。

24.1　节点类型

所有的Spout都会放在一个Spout节点中运行，不会与其他运算单元合并。Topology的一个Bolt节点中，可能还有多个处理节点，每个处理节点对应于用户定义的操作。Spout节点与Bolt节点之间利用分区节点进行连接，它对消息的分组方式进行抽象。Trident中节点类型间的关系如图24-1所示。

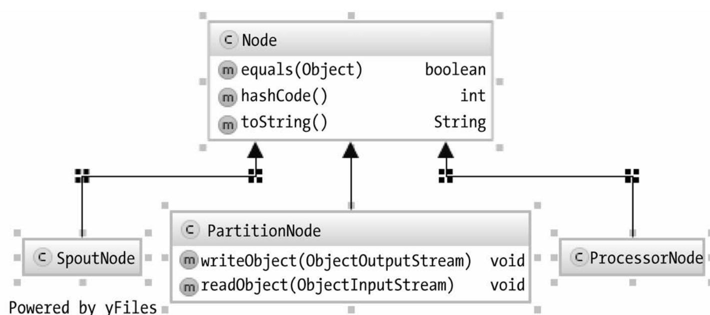


图24-1　Trident中的节点类型

24.1.1　基本节点类型

Node类为Topology所对应的有向图中的节点，理解其成员变量是重要的，该类的代码如下：

```
public class Node implements Serializable {
    private static AtomicInteger INDEX = new AtomicInteger(0);

    private String nodeId;
```

```
    public String name = null;
    public Fields allOutputFields;
    public String streamId;
    public Integer parallelismHint = null;
    public NodeStateInfo stateInfo = null;
    public int creationIndex;

    public Node(String streamId, String name, Fields allOutputFields) {
        this.nodeId = UUID.randomUUID().toString();
        this.allOutputFields = allOutputFields;
        this.streamId = streamId;
        this.name = name;
        this.creationIndex = INDEX.incrementAndGet();
}
```

- INDEX：用来给Topology中的每一个节点创建一个编号。
- creationIndex：利用INDEX创建的编号。这个编号反应了该节点在Topology中的层次关系。在对图进行拓扑排序时，编号大的节点处于拓扑序列的后面。例如，DRPC需要创建一个节点来向DRPC服务器发送结果，Trident利用creationIndex来找到该节点的父节点，这个父节点为DRPC子图中编号最大的节点（未创建DRPC节点之前）。
- nodeId：一个节点的全局唯一标识符。
- name：节点名字。该名字会显示在UI上面，由于Trident创建了较多的隐藏节点，目前对名字的支持并不好，通常name域均为空。
- parallelismHint：表示该节点的并行度。同一节点组中的不同节点，其并行度可能会不一致，通常会取节点组中最大parallelismHint作为整个节点组的并行度。
- stateInfo：表示该节点是否会进行存储。类NodeStateInfo的定义如下，其中id为全局唯一的State标识符，StateSpec则规定了如何去创建一个存储对象，以及该存储对象是否要求有固定的分区数目。

```
public class NodeStateInfo implements Serializable {
    public String id;
    public StateSpec spec;
    public NodeStateInfo(String id, StateSpec spec) {
        this.id = id;
        this.spec = spec;
    }
}
```

目前，主要有3种类型的节点。

- SpoutNode：Spout节点，进一步区分为DRPC Spout类型和基本Spout类型。
- ProcessorNode：处理节点。该类型的节点会包含实际的运算单元，是构成SubTopologyBolt的主要部分。
- PartitionNode：分区节点。该类型节点只适用于描述节点的输出会如何被接收（例如某

个节点的流将以何种方式被其他节点处理等）。分区节点对应于Storm中的消息分组方式。分区节点是Topology执行优化中最重要的部分，它是节点组之间的分割节点。

24.1.2　Spout节点

Spout节点的实现代码如下：

```
public class SpoutNode extends Node {
    public static enum SpoutType {
        DRPC,
        BATCH
    }

    public Object spout;
    public String txId; //where state is stored in zookeeper (only for batch spout types)
    public SpoutType type;

    public SpoutNode(String streamId, Fields allOutputFields, String txid, Object spout, SpoutType type) {
        super(streamId, null, allOutputFields);
        this.txId = txid;
        this.spout = spout;
        this.type = type;
    }
}
```

- _type成员变量表明该Spout是否为DRPC Spout节点。DRPC Spout是较为特殊的，它的每个事务都只包含一条消息。
- txId成员变量对应于Spout节点的名称。Trident Spout中的协调Spout需要在ZooKeeper中存储一些元数据，txId将作为在ZooKeeper中元数据路径的一部分。

24.1.3　处理节点

处理节点的实现分析如下：

```
public class ProcessorNode extends Node {

    public boolean committer; // for partitionpersist
    public TridentProcessor processor;
    public Fields selfOutFields;

    public ProcessorNode(String streamId, String name, Fields allOutputFields, Fields selfOutFields,
        TridentProcessor processor) {
        super(streamId, name, allOutputFields);
        this.processor = processor;
        this.selfOutFields = selfOutFields;
    }
}
```

- _processor：TridentProcessor类型的成员变量，它包含了真正的计算单元。

- _selfOutFields：该节点将自己产生的新列。
- _committer：一个布尔类型的值，处理节点在进行partitionPersist操作时将其设为真。由于partitionPersist通常会向存储对象中更新数据，那么何时为合适的更新时机呢？_committer变量的作用在于，它可以告知Topology的构建器，该节点需要收听从协调Spout那里发送出来的事务提交流。在收到某一个事务的提交消息时，节点便得知该事务已经被成功处理了，也就获得了一个合适的更新时机。

24.1.4 分区节点

分区节点的分析如下：

```
public class PartitionNode extends Node {
    public transient Grouping thriftGrouping;

    //has the streamid/outputFields of the node it's doing the partitioning on
    public PartitionNode(String streamId, String name, Fields allOutputFields, Grouping grouping) {
        super(streamId, name, allOutputFields);
        this.thriftGrouping = grouping;
    }

    private void writeObject(ObjectOutputStream oos) throws IOException {
        oos.defaultWriteObject();
        byte[] ser = TridentUtils.thriftSerialize(thriftGrouping);
        oos.writeInt(ser.length);
        oos.write(ser);
    }

    private void readObject(ObjectInputStream ois) throws ClassNotFoundException, IOException {
        ois.defaultReadObject();
        byte[] ser = new byte[ois.readInt()];
        ois.readFully(ser);
        this.thriftGrouping = TridentUtils.thriftDeserialize(Grouping.class, ser);
    }
}
```

- thriftGrouping是分区节点新增加的成员变量。
- 它是通过Thrift定义文件产生的Grouping类型，所以会采用Thrift的序列化器来实现序列化和反序列化方法，并且覆盖了默认的序列化和反序列方法。

在Trident中，可实现几种定制的分组方式来协助Topology的执行优化以及使编程更加方便，现统一介绍如下。

IdentityGrouping类用来表示在源端组件的Task数目与接收端组件的Task数目相同的情况下，源端的Task与目的端的Task可以一一对应起来，这与直接分组是非常类似的。

在Topology的执行优化过程中，如果两个相邻的节点组通过IdentityGrouping的方式进行了连接，则表示它们具有相同的并行度，于是会将这两个节点组合并。该类的分析如下：

```
1 public class IdentityGrouping implements CustomStreamGrouping {
2
3     List<Integer> ret = new ArrayList<Integer>();
4     Map<Integer, List<Integer>> _precomputed = new HashMap();
5
6     @Override
7     public void prepare(WorkerTopologyContext context, GlobalStreamId stream,
          List<Integer> tasks) {
8         List<Integer> sourceTasks = new ArrayList<Integer>(context.getComponentTasks(stream
            .get_componentId()));
9         Collections.sort(sourceTasks);
10        if(sourceTasks.size()!=tasks.size()) {
11            throw new RuntimeException("Can only do an identity grouping when
                  source and target have same number of tasks");
12        }
13        tasks = new ArrayList<Integer>(tasks);
14        Collections.sort(tasks);
15        for(int i=0; i<sourceTasks.size(); i++) {
16            int s = sourceTasks.get(i);
17            int t = tasks.get(i);
18            _precomputed.put(s, Arrays.asList(t));
19        }
20    }
21
22    @Override
23    public List<Integer> chooseTasks(int task, List<Object> values) {
24        List<Integer> ret = _precomputed.get(task);
25        if(ret==null) {
26            throw new RuntimeException("Tuple emitted by task that's not part
                  of this component. Should be impossible");
27        }
28        return ret;
29    }
30 }
```

- 定制的分组方式需要实现CustomStreamGrouping接口。具体地讲，就是在prepare方法中需要传入接收端的TaskId列表，在chooseTasks方法中需根据当前Task及当前消息内容来选择接收端的一个或多个节点。
- ret为类成员变量，它并没有被用到，可以删除。
- _precomputed用来存储预先定义好的源Task节点与目标Task节点的映射关系。由于chooseTasks要求返回一个列表类型，因此虽然_precomputed中只包含一个节点，但它仍被定义为列表类型。
- 第7~20行实现prepare方法。第8~9行获得所有源端节点的Task节点并进行排序，第10~12行要求源端以及目标端的Task数目相同，否则便不适用于这种分组方式了。第15~19行构建源端与目标端Task的一一对应关系，并存储于_precomputed类成员变量中。
- 第23~29行实现chooseTasks方法。Trident会在源端产生并向外发送一条消息时会调用该方法，其中参数task表示源端的TaskId，参数values表示消息的内容。IdentityGrouping并不需要利用消息的内容，而只需要根据源端的TaskId找到预先计算好的目的端TaskId即可。

类IndexHashGrouping实现的功能与域分组非常类似，区别在于域分组需要传入Fields对象（即字段名）来进行分组，而IndexHashGrouping则根据某一个列的下标来进行分组。例如在DRPC处理中我们知道，第1列为事务序号列，且向DRPC发送结果的节点需要按照事务序号进行连接。于是Trident便可以使用IndexHashGrouping分组方式并传入下标0，这样只要保证事务序号相同消息就会到达相同的Bolt节点，在此基础上便可以执行连接操作。IndexHashGrouping的代码如下：

```
1 public class IndexHashGrouping implements CustomStreamGrouping {
2     public static int objectToIndex(Object val, int numPartitions) {
3         if(val==null) return 0;
4         else {
5             return Math.abs(val.hashCode()) % numPartitions;
6         }
7     }
8
9     int _index;
10    List<Integer> _targets;
11
12    public IndexHashGrouping(int index) {
13        _index = index;
14    }
15
16
17    @Override
18    public void prepare(WorkerTopologyContext context, GlobalStreamId stream, List<Integer>
          targetTasks) {
19        _targets = targetTasks;
20    }
21
22    @Override
23    public List<Integer> chooseTasks(int fromTask, List<Object> values) {
24        int i = objectToIndex(values.get(_index), _targets.size());
25        return Arrays.asList(_targets.get(i));
26    }
27 }
```

- 第2~7行，objectToIndex函数会根据列内容及目标端的Task数目，以哈希方式选择目标TaskId。
- _index成员变量用来指明依照哪一列来选择目标TaskId。
- 第18~20行的prepare方法将获取目的端的Task，此处可以只存储数目。
- 第23~27行的chooseTasks方法会从输入消息values中获取第_index列元素，并调用objectToIndex来得到目标TaskId。

GlobalGrouping类将所有由源端Task产生的消息都发送到一个特定的目的端Task，这与全局分组所实现的功能相同。该类的代码如下：

```
public class GlobalGrouping implements CustomStreamGrouping {

    List<Integer> target;
```

```
    @Override
    public void prepare(WorkerTopologyContext context, GlobalStreamId stream, List<Integer> targets) {
        List<Integer> sorted = new ArrayList<Integer>(targets);
        Collections.sort(sorted);
        target = Arrays.asList(sorted.get(0));
    }

    @Override
    public List<Integer> chooseTasks(int i, List<Object> list) {
        return target;
    }
}
```

GlobalGrouping会选取排序后的TaskId列表中，第一个目的端TaskId作为目标节点。

24.2　执行优化算法

Trident引入节点组的概念，属于同一个节点组的节点最终将被放入同一个Bolt节点中执行。Trident会采用一些图的算法来试图将节点组合并，使得尽可能多的操作在同一个Bolt节点中完成。

本节将分析如何产生节点组以及节点组的合并算法。

24.2.1　节点组

节点组是构建SubTopologyBolt的基础，也是Topology中执行优化的基本操作单元，Trident会通过不断地合并节点组来达到最优处理的目的。用户可使用类Group来创建节点组，其中包含了一组连通的节点。该类的代码如下：

```
1 public class Group {
2     public Set<Node> nodes = new HashSet<Node>();
3     private DirectedGraph<Node, IndexedEdge> graph;
4     private String id;
5
6     public Group(DirectedGraph graph, List<Node> nodes) {
7         init(graph);
8         this.nodes.addAll(nodes);
9         this.graph = graph;
10    }
11
12     public Group(DirectedGraph graph, Node n) {
13         this(graph, Arrays.asList(n));
14     }
15
16     public Group(Group g1, Group g2) {
17         init(g1.graph);
18         nodes.addAll(g1.nodes);
19         nodes.addAll(g2.nodes);
20     }
```

```
21
22     private void init(DirectedGraph graph) {
23         this.graph = graph;
24         this.id = UUID.randomUUID().toString();
25     }
26
27     public Set<Node> outgoingNodes() {
28         Set<Node> ret = new HashSet<Node>();
29         for(Node n: nodes) {
30             ret.addAll(TridentUtils.getChildren(graph, n));
31         }
32         return ret;
33     }
34
35     public Set<Node> incomingNodes() {
36         Set<Node> ret = new HashSet<Node>();
37         for(Node n: nodes) {
38             ret.addAll(TridentUtils.getParents(graph, n));
39         }
40         return ret;
41     }
42 }
```

- nodes成员变量表示该节点组中含有的节点。
- graph成员变量表示Topology节点的有向图。
- id为GUID类型，用以唯一地标识一个节点组。
- 第27~33行定义的outgoingNodes方法，会通过遍历组中节点的方式来获取该节点组所有节点的子节点，这些子节点可能属于该节点组，也可能属于其他的节点组。
- 第35~41行定义的incomingNodes方法用来获得该节点组中所有节点的父节点，这些父节点可能属于该节点组，也可能属于其他的节点组。
- 第16~17行的构造函数用于合并两个节点组g1和g2。第12~14行定义的构造函数则用于创建只含一个节点的节点组。Topology在进行执行优化时，最开始的状态即是这种每一个处理节点独立作为一个节点组的情况。
- 第6~10行的构造函数用来对节点组进行初始化。其中， init函数的作用仅为创建节点组id，以及对graph赋值。

24.2.2 节点组的合并算法

类GraphGrouper提供了对节点组进行操作及合并的基本算法：

```
public class GraphGrouper {
    DirectedGraph<Node, IndexedEdge> graph;
    Set<Group> currGroups;
    Map<Node, Group> groupIndex = new HashMap();
}
```

- graph成员变量的意义与前面节点组中的相同，表示Topology中节点的有向图。

- currGroups表示当前graph所对应的节点组。节点组之间是没有交集的。
- groupIndex是一个反向的索引，用于快速查询每个节点所在的节点组。该成员变量在节点组的合并操作中起到了重要作用。

下面对GraphGrouper中的重要工具函数进行介绍。

reindex函数会根据currGroups来重新构建groupIndex索引。其代码如下：

```
public void reindex() {
    groupIndex.clear();
    for(Group g: currGroups) {
        for(Node n: g.nodes) {
            groupIndex.put(n, g);
        }
    }
}
```

nodeGroup函数用于查询某一个节点所在的节点组：

```
public Group nodeGroup(Node n) {
    return groupIndex.get(n);
}
```

工具方法outgoingGroups会根据输入节点组和有向图来计算：哪些节点组与输入的节点组之间存在有向边，即两个节点组是相连的。其基本算法为遍历每一个节点的子节点，若该子节点所在的节点组与自身节点组不同，则获得子节点所在的节点组。该方法的代码如下：

```
public Collection<Group> outgoingGroups(Group g) {
    Set<Group> ret = new HashSet();
    for(Node n: g.outgoingNodes()) {
        Group other = nodeGroup(n);
        if(other==null || !other.equals(g)) {
            ret.add(other);
        }
    }
    return ret;
}
```

incomingGroups方法与outgoingGroups相似，用来获得该节点组的父节点组，即父节点组中至少存在一个节点及一条有向边与该节点组中的节点相连。其代码如下：

```
public Collection<Group> incomingGroups(Group g) {
    Set<Group> ret = new HashSet();
    for(Node n: g.incomingNodes()) {
        Group other = nodeGroup(n);
        if(other==null || !other.equals(g)) {
            ret.add(other);
        }
    }
    return ret;
}
```

注意变量other是可能为空的，因为在合并节点组的算法中，并不是每次都会重新更新索引。

merge函数用来合并两个节点组。算法为将g1和g2从currGroups中去除，同时添加一个新的节点组，该节点组包含g1和g2的所有节点，最后将groupIndex更新到新的节点组。函数的代码如下：

```
private void merge(Group g1, Group g2) {
    Group newGroup = new Group(g1, g2);
    currGroups.remove(g1);
    currGroups.remove(g2);
    currGroups.add(newGroup);
    for(Node n: newGroup.nodes) {
        groupIndex.put(n, newGroup);
    }
}
```

mergeFully方法是GraphGrouper的核心算法，它用来计算何时可以对两个节点组进行合并。其基本思想为：如果一个节点组只有一个父节点组，那么将该节点组与其父节点组进行合并；类似地，如果一个节点组只有一个子节点组，那么将子节点组与自身节点组进行合并；反复进行这两个过程直到没有满足以上条件的节点组为止。mergeFully方法的代码如下：

```
1 public void mergeFully() {
2     boolean somethingHappened = true;
3     while(somethingHappened) {
4         somethingHappened = false;
5         for(Group g: currGroups) {
6             Collection<Group> outgoingGroups = outgoingGroups(g);
7             if(outgoingGroups.size()==1) {
8                 Group out = outgoingGroups.iterator().next();
9                 if(out!=null) {
10                    merge(g, out);
11                    somethingHappened = true;
12                    break;
13                }
14            }
15
16            Collection<Group> incomingGroups = incomingGroups(g);
17            if(incomingGroups.size()==1) {
18                Group in = incomingGroups.iterator().next();
19                if(in!=null) {
20                    merge(g, in);
21                    somethingHappened = true;
22                    break;
23                }
24            }
25        }
26    }
27 }
```

- 第2行的变量somethingHappened用来表示当前节点组只有一个父或子节点组，并且已经进行了合并操作，因此需要进行下一次迭代。
- 第6~14行首先获得一个节点组的所有父节点组，若父节点组的数目为1，则进行合并。

- 类似地，第16~25行处理只有一个子节点组的情况。

该类的构造函数需要传入Topology所对应的有向图，以及初始节点组initialGroups。函数的代码如下：

```
public GraphGrouper(DirectedGraph<Node, IndexedEdge> graph, Collection<Group> initialGroups) {
    this.graph = graph;
    this.currGroups = new HashSet(initialGroups);
    reindex();
}
```

理解初始化节点组initialGroups的获取过程是非常重要的。为了达到这个目的，我们来分析下面这段代码，它摘自于文件TridentTopology.java：

```
1  List<SpoutNode> spoutNodes = new ArrayList<SpoutNode>();
2  Set<Node> boltNodes = new HashSet<Node>();
3  for(Node n: graph.vertexSet()) {
4      if(n instanceof SpoutNode) {
5          spoutNodes.add((SpoutNode) n);
6      } else if(!(n instanceof PartitionNode)) {
7          boltNodes.add(n);
8      }
9  }
10
11 Set<Group> initialGroups = new HashSet<Group>();
12 for(List<Node> colocate: _colocate.values()) {
13     Group g = new Group(graph, colocate);
14     boltNodes.removeAll(colocate);
15     initialGroups.add(g);
16 }
17 for(Node n: boltNodes) {
18     initialGroups.add(new Group(graph, n));
19 }
20
21 GraphGrouper grouper = new GraphGrouper(graph, initialGroups);
```

- 第2~9行遍历了有向图中的所有节点，并将它们区分为Spout节点或Bolt节点。注意，分区节点不属于Bolt节点。
- 第17~19行将每一个Bolt节点初始化为一个节点组，这表明分区节点并不属于任何的节点组。分区节点起到了连接节点组的作用，当它出现时，与它连接的节点所在的节点组是不可能被合并的。
- 第11~16行将_colocate变量中的节点组初始化为同一个节点组。_colocate变量是一个哈希表，其键为StateId，值为该State上进行操作的节点（即对State进行更新和查询的节点）。Trident会将这两类节点放在一个节点组中同时处理，以便于State的查询与更新。 可以想象，如果不把这两类节点放在同一个节点组中，查询操作将很难完成。例如，在含有DRPC节点的Topology中，进行StateQuery操作的节点与PartitionPersist所对应的节点将会被放在同一个节点组，也即同一个Bolt中执行。由于这两类节点可能属于不同的节点组，这也为TridentBoltExecutor增加了不少的复杂性。

- 于是可以总结：分区节点起到分割节点组的作用；Trident会尽量地将节点组合并，合并后的节点组将在同一个Bolt中执行。

24.2.3 处理节点组中的分区节点

为了处理方便，Trident要求节点组之间一定要用分区节点来进行连接。当然，节点组与Spout节点组之间不需要满足此要求。

下面的代码段用于进行异常情况处理：

```
1 for(IndexedEdge<Node> e: new HashSet<IndexedEdge>(graph.edgeSet())) {
2     if(!(e.source instanceof PartitionNode) && !(e.target instanceof PartitionNode)) {
3         Group g1 = grouper.nodeGroup(e.source);
4         Group g2 = grouper.nodeGroup(e.target);
5         // g1 being null means the source is a spout node
6         if(g1==null && !(e.source instanceof SpoutNode))
7         throw new RuntimeException("Planner exception: Null source group must indicate a
              spout node at this phase of planning");
8          if(g1==null || !g1.equals(g2)) {
9              graph.removeEdge(e);
10             PartitionNode pNode = makeIdentityPartition(e.source);
11             graph.addVertex(pNode);
12             graph.addEdge(e.source, pNode, new IndexedEdge(e.source, pNode, 0));
13             graph.addEdge(pNode, e.target, new IndexedEdge(pNode, e.target, e.index));
14         }
15     }
16 }
17
18 private static PartitionNode makeIdentityPartition(Node basis) {
19     return new PartitionNode(basis.streamId, basis.name, basis.allOutputFields,
20         Grouping.custom_serialized(Utils.serialize(new IdentityGrouping())));
21 }
```

- 第18~21行的`makeIdentityParition`函数创建了一个新的分区节点，它与父节点之间按照`IdentityGrouping`的方式进行连接。根据前面的讨论，`IdentityGrouping`要求源节点与目的节点之间具有相同的Task数目。
- 第1~16行对有向图中所有的边进行遍历，获得每一条边连接的节点所在的节点组。第6~7行表明，若源端节点组为空并且其节点也不是Spout节点，则处于异常状态。若g1与g2所在的节点组不同，则创建一个新的分区节点，并将该分区节点插入到源节点与目的节点之间。
- 通过这段代码，Trident保证了Bolt节点组之间均通过分区节点完成连接。

24

24.2.4 节点组以不同的方式收听相同流

Storm本身并不支持一个节点组中的节点以不同的分组方式去收听同一个流。对于这种情况，Trident会创建一个新的等同节点，以及一个新的分区节点，使得每个输入的分区节点所对应的流只存在一种分组方式。

externalGroupInputs方法用于获得一个节点组的所有父分区节点。根据前面的分析，同一个节点组中，可以含有分区节点，但这些分区节点不会含有子节点。Bolt节点组之间通过分区节点进行连接，故externalGroupInputs获得的是一个节点组的所有外部输入。注意，incomingNodes方法可以返回一个节点组内部的节点。externalGroupInputs方法的代码如下：

```
private static Set<PartitionNode> externalGroupInputs(Group g) {
    Set<PartitionNode> ret = new HashSet();
    for(Node n: g.incomingNodes()) {
        if(n instanceof PartitionNode) {
            ret.add((PartitionNode) n);
        }
    }
    return ret;
}
```

extraPartitionInputs方法用来计算得到同一流的具有不同分组方式的分区节点，其代码如下：

```
1  private static List<PartitionNode> extraPartitionInputs(Group g) {
2      List<PartitionNode> ret = new ArrayList();
3      Set<PartitionNode> inputs = externalGroupInputs(g);
4      Map<String, List<PartitionNode>> grouped = new HashMap();
5      for(PartitionNode n: inputs) {
6          if(!grouped.containsKey(n.streamId)) {
7              grouped.put(n.streamId, new ArrayList());
8          }
9          grouped.get(n.streamId).add(n);
10     }
11     for(List<PartitionNode> group: grouped.values()) {
12         PartitionNode anchor = group.get(0);
13         for(int i=1; i<group.size(); i++) {
14             PartitionNode n = group.get(i);
15             if(!n.thriftGrouping.equals(anchor.thriftGrouping)) {
16                 ret.add(n);
17             }
18         }
19     }
20     return ret;
21 }
```

- 第2行调用externalGroupInputs方法来获得所有的外部输入分区节点。
- 第4~10行根据流的名字对这些分区节点进行分组，若同一个流具有多个分区节点，且分组方式不同，则将这些节点返回。
- 第13行的变量i是从1开始的。正如其名称所表达的意思，函数返回的为额外的分区输入节点。若一个流的所有分区节点拥有相同的分组方式，Trident则不需要进行额外处理。

下面的代码完成了“节点组以不同的方式收听同一流”这一特殊情况的处理逻辑。另外需注意，代码中的注释与实际的逻辑并不符合：

```
1 // if one group subscribes to the same stream with same partitioning multiple times,
2 // merge those together (otherwise can end up with many output streams created for that
        partitioning
3 // if need to split into multiple output streams because of same input having different
4 // partitioning to the group)
5
6 // this is because can't currently merge splitting logic into a spout
7 // not the most kosher algorithm here, since the grouper indexes are being trounced via
        the adding of nodes to random groups, but it
8 // works out
9 List<Node> forNewGroups = new ArrayList<Node>();
10    for(Group g: mergedGroups) {
11    for(PartitionNode n: extraPartitionInputs(g)) {
12        Node idNode = makeIdentityNode(n.allOutputFields);
13        Node newPartitionNode = new PartitionNode(idNode.streamId, n.name,
             idNode.allOutputFields, n.thriftGrouping);
14        Node parentNode = TridentUtils.getParent(graph, n);
15        Set<IndexedEdge> outgoing = graph.outgoingEdgesOf(n);
16        graph.removeVertex(n);
17
18        graph.addVertex(idNode);
19        graph.addVertex(newPartitionNode);
20        addEdge(graph, parentNode, idNode, 0);
21        addEdge(graph, idNode, newPartitionNode, 0);
22        for(IndexedEdge e: outgoing) {
23            addEdge(graph, newPartitionNode, e.target, e.index);
24        }
25        Group parentGroup = grouper.nodeGroup(parentNode);
26        if(parentGroup==null) {
27            forNewGroups.add(idNode);
28         } else {
29             parentGroup.nodes.add(idNode);
30         }
31     }
32 }
33
34 for(Node n: forNewGroups) {
35     grouper.addGroup(new Group(graph, n));
36 }
37
38 // add in spouts as groups so we can get parallelisms
39 for(Node n: spoutNodes) {
40     grouper.addGroup(new Group(graph, n));
41 }
```

- 第10~32行代码对每一个节点组进行处理，第11行对于每一个额外的分区节点进行处理。
- 第12行创建了一个相同的节点，该节点中含有一个EachProcessor。EachProcessor利用TrueFilter将输入的消息原封不动地输出，因此被称为等同节点。第13行创建一个新的分区节点，该节点的输入为新创建的等同节点。
- 第16~24行对有向图进行更新。删除掉老的节点，插入新的节点，并更新边的关系和节点组的关系。

- 第34~36行将新产生的节点设置为新的节点组。
- 第39~41行将Spout 节点作为节点组，即Spout 节点与Bolt 的节点组之间并不进行合并。

24.2.5　执行优化后的节点组

完成了以上的步骤，也就基本上完成了对有向图中节点组的优化。下面的代码将重新计算节点所属的节点组，并运行节点组的合并算法得到mergedGroups。然后，利用图论中的最大连通子图算法获得connectedComponents变量，称这样的一个最大连通子图为一个节点组。

Topology所对应的有向图中，任何一个节点只能属于某一个节点组。Bolt组件上可以运行属于不同节点组的节点。例如，在含有DRPC的Topology中，某个SubTopologyBolt可以运行DRPC 的查询操作，以及数据的存储操作，这两个操作就属于不同的节点组。

```
grouper.reindex();
mergedGroups = grouper.getAllGroups();


Map<Node, String> batchGroupMap = new HashMap();
List<Set<Node>> connectedComponents = new ConnectivityInspector<Node, IndexedEdge>(graph).
    connectedSets();
for(int i=0; i<connectedComponents.size(); i++) {
    String groupId = "bg" + i;
    for(Node n: connectedComponents.get(i)) {
        batchGroupMap.put(n, groupId);
    }
}
```

24.2.6　计算节点组的并行度

属于同一个节点组的节点会在同一个Bolt 中执行，那么这个Bolt 的并行度该如何得到呢？Trident进一步使用了图的算法，它利用节点组作为图的顶点，当节点组之间通过分组方式为IdentityGrouping的分区节点进行连接时，就将这两个节点组进行合并。然后获得最大连通子图，并在每个连通子图上计算并行度，最终便得到了每一个节点组的并行度。相关的代码及方法分析如下：

```
1 private static Map<Group, Integer> getGroupParallelisms(DirectedGraph<Node,
      IndexedEdge> graph, GraphGrouper grouper, Collection<Group> groups) {
2     UndirectedGraph<Group, Object> equivs = new Pseudograph<Group, Object>(Object.class);
3     for(Group g: groups) {
4         equivs.addVertex(g);
5     }
6     for(Group g: groups) {
7         for(PartitionNode n: externalGroupInputs(g)) {
8             if(isIdentityPartition(n)) {
9                 Node parent = TridentUtils.getParent(graph, n);
10                Group parentGroup = grouper.nodeGroup(parent);
11                if(parentGroup!=null && !parentGroup.equals(g)) {
12                    equivs.addEdge(parentGroup, g);
```

```
13                }
14            }
15        }
16    }
17
18    Map<Group, Integer> ret = new HashMap();
19    List<Set<Group>> equivGroups = new ConnectivityInspector<Group, Object>(equivs)
         .connectedSets();
20    for(Set<Group> equivGroup: equivGroups) {
21        Integer fixedP = getFixedParallelism(equivGroup);
22        Integer maxP = getMaxParallelism(equivGroup);
23        if(fixedP!=null && maxP!=null && maxP < fixedP) {
24           throw new RuntimeException("Parallelism is fixed to " + fixedP + " but max parallelism
          is less than that: " + maxP);
25        }
26
27
28        Integer p = 1;
29        for(Group g: equivGroup) {
30            for(Node n: g.nodes) {
31                if(n.parallelismHint!=null) {
32                    p = Math.max(p, n.parallelismHint);
33                }
34            }
35        }
36        if(maxP!=null) p = Math.min(maxP, p);
37
38        if(fixedP!=null) p = fixedP;
39        for(Group g: equivGroup) {
40            ret.put(g, p);
41        }
42    }
43    return ret;
44 }
```

- 第2行生成一个Pseudograph对象，它是一种可以在两个节点之间添加多条边的图。
- 第3~5行将节点组作为equivs图的顶点。
- 第6~16行当节点组之间的分组方式为IdentityGrouping时，则创建一条边，表示这两个顶点所对应的节点组将具有相同的并行度。
- 第18~19行获得equivs的最大连通子图。
- 第21行获得固定并行度。StateInfo对象可以指定分区数目，表示它代表的State对象存在多少个分区。Trident认为节点计算的并行度最好与State的分区数目一致。固定并行度通过节点组中含有State的节点的分区数目计算得到，同一个节点组中不允许存在两个节点均含有State对象的情况，但其分区数目可以不一致的情况。当没有分区的State时，固定并行度为空。
- 第22行计算最大并行度。最大并行度只对含有Spout的节点组有效，其他类型的节点组为空。它是根据Spout节点中配置项TOPOLOGY_MAX_TASK_PARALLELISM的设定来获得的。因此，固定并行度一定要小于等于最大并行度。

- 第28~35行计算初步的并行度，默认的并行度为1。取节点中所有节点的parallelismHint的最大值作为节点组的并行度。理解如何产生节点组对于设置并行度是重要的，用户设置的parallelismHint仅作为产生并行度的参考而非是最终的并行度。
- 第36行表示若计算得到的并行度大于最大并行度，则取最大并行度为节点组的并行度。
- 第38行表示若固定并行度不为空，取固定并行度为节点组的并行度。
- 第39~41行将节点组的并行度存储在结果中以便返回。

第 25 章

Trident与DRPC

DRPC是英文Distributed Remote Procedure Call的缩写，表示分布式的远程方法调用，它是Trident中重要的功能模块，主要用于支持快速的分布式查询操作。你可访问下面的链接来获得有关DRPC的更多内容：https://github.com/nathanmarz/storm/wiki/Distributed-RPC。

DRPC的总体结构如图25-1所示（该图源自官方文档）。

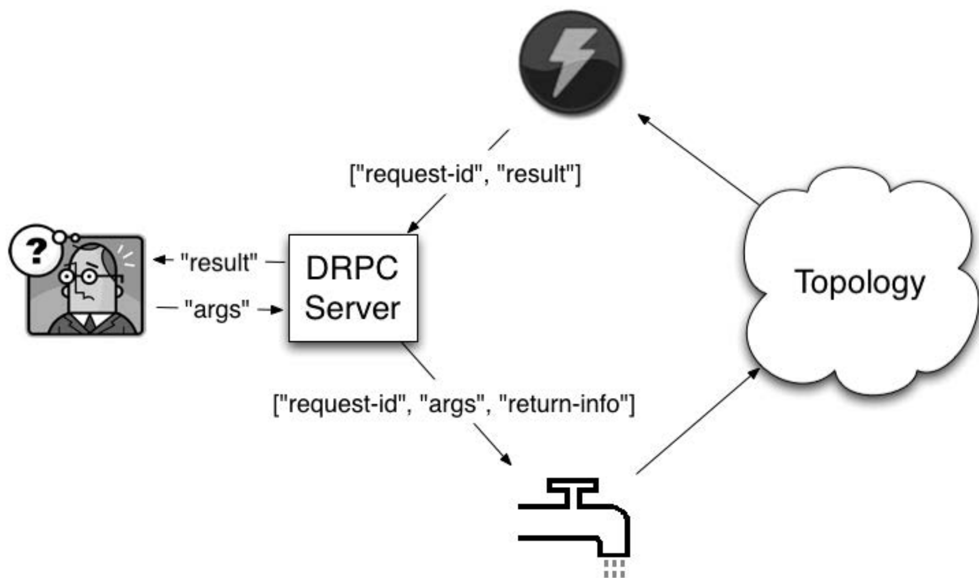


图25-1　DRPC的总体架构

DRPC Server是独立于Topology的DRPC服务器，它接收用户的输入，同时也作为DRPC Spout的输入而存在。DRPC Spout从DRPC服务器获得要执行的查询请求，并在Topology中执行计算。最终，Topology中的某个Bolt会将结果发送给DRPC服务器，再由DRPC服务器返回给用户。

DRPC请求通常以较快的速度返回。该请求通常已经被编译成为Topology中的一部分，并与其他的节点一并提交、运行，只不过DRPC Spout只有当存在输入请求时才会有消息发送给Topology。

本章将对DRPC的各个部分进行介绍。

25.1　DRPC 服务器

DRPC服务器包含两部分服务接口：一部分用于接收用户请求并传递结果，另一部分则用于向DRPC Spout提交请求并获得Bolt节点的计算结果。本节对DRPC服务器进行讨论。

25.1.1　DRPC服务器的成员变量

首先来分析DRPC服务器的数据，其代码如下：

```
1 (let [conf (read-storm-config)
2         ctr (atom 0)
3         id->sem (atom {})
4         id->result (atom {})
5         id->start (atom {})
6         request-queues (atom {})
7         cleanup (fn [id] (swap! id->sem dissoc id)
8                     (swap! id->result dissoc id)
9                     (swap! id->start dissoc id))
10        my-ip (.getHostAddress (InetAddress/getLocalHost))
11        clear-thread (async-loop
12                       (fn []
13                           (doseq [[id start] @id->start]
14                              (when (> (time-delta start) (conf DRPC-REQUEST-TIMEOUT-SECS))
15                                  (when-let [sem (@id->sem id)]
16                                      (swap! id->result assoc id (DRPCExecutionException.
                                            "Request timed out"))
17                                      (.release sem))
18                                  (cleanup id)
19                                  ))
20                           TIMEOUT-CHECK-SECS
21                           ))
22      ])
```

- ctr为服务器端的计数器，用来为输入的请求产生一个Id。它为atom类型变量。atom非常适合实现缓存，缓存通常不会跟其他系统状态形成依赖，并且对读取的速度要求更高。如果你有一个状态变量，且不需要跟其他状态变量协作，这时候就应该使用atom。
- id->sem为atom类型的哈希表，用来存储从RequestId到Semaphore变量的映射关系，Semophore（信号量）可用于不同线程间的同步。DRPC服务器为每一个请求创建了一个信号量，因此，目前其实现负担是较大的。
- id->result为atom类型的哈希表，用来存储从RequestId到结果的映射关系。
- id->start为atom类型的哈希表，用来存储从RequestId到该请求收到的时间的映射关系。
- cleanup是一个成员函数，用于在请求处理结束后清理缓存，主要清理id->sem、id->result和id->start缓存。
- clear-thread是服务器端的一个独立线程，它会遍历id->start哈希表，并根据该请求的启动时间和DRPC-REQUEST-TIMEOUT-SECS定义的时间判断请求是否已经超时。如果超时，则

将一个异常放在id->result哈希表中，同时调用信号量的释放操作。这样，等待结果的线程就会收到这个通知，并发现这一信息为异常，最终将这一信息返回给用户。

- TIMEOUT-CHECK-SECS变量定义了clear-thread执行的时间间隔。
- clear-thread保证了系统资源的释放，以及用户在规定时间内可以得到响应。
- request-queues为atom类型的哈希表，它的键为函数名称，值为一个队列，表示对应于该函数的所有请求。

下面的方法将返回一个队列：

```
1 (defn acquire-queue [queues-atom function]
2   (swap! queues-atom
3     (fn [amap]
4       (if-not (amap function)
5         (assoc amap function (ConcurrentLinkedQueue.))
6         amap)
7         ))
8   (@queues-atom function))
```

- swap!将对输入的atom类型的哈希表进行操作，其结果为fn执行的结果。
- 第4行，若amap不含有键funciton，则创建一个ConcurrentLinkedQueue并与函数名称相关联。
- 第8行返回ConcurrentLinkedQueue，表示输入为函数的请求队列。

25.1.2 DRPC用户接口及其实现

这部分接口主要提供给用户使用，用于向DRPC服务器提交查询并等待返回。该接口的代码如下：

```
1 (reify DistributedRPC$Iface
2    (^String execute [this ^String function ^String args]
3       (log-debug "Received DRPC request for " function " " args " at " (System/
          currentTimeMillis))
4       (let [id (str (swap! ctr (fn [v] (mod (inc v) 1000000000))))
5                ^Semaphore sem (Semaphore. 0)
6                req (DRPCRequest. args id)
7                ^ConcurrentLinkedQueue queue (acquire-queue request-queues function)
8                ]
9            (swap! id->start assoc id (current-time-secs))
10           (swap! id->sem assoc id sem)
11           (.add queue req)
12           (log-debug "Waiting for DRPC result for " function " " args " at " (System/
                currentTimeMillis))
13           (.acquire sem)
14           (log-debug "Acquired DRPC result for " function " " args " at " (System/
                currentTimeMillis))
15           (let [result (@id->result id)]
16                (cleanup id)
17                (log-debug "Returning DRPC result for " function " " args " at " (System/
                    currentTimeMillis))
```

```
18                (if (instance? DRPCExecutionException result)
19                    (throw result)
20                    result
21                    ))))
```

- DRPC面向用户的部分只有execute方法，它的参数为函数名字，以及该函数所对应的参数。在Topology中，每个函数名都会对应于一个DRPC Spout。
- 第4行，产生该请求的序号RequestId。它通过对ctr对象进行自增并模除1000000000得到。
- 第5行产生一个信号量。
- 第6行构建DRPCRequest对象。
- 第7行获得与请求函数相对应的请求队列。
- 第9~11行更新id->start和id->sem缓存，同时将请求req放入队列。
- 第13行开始等待信号量sem的释放。只有当结果返回或者超时时，第13行才会返回内容。
- 第15~21行对返回的结果进行处理，若为DRPCExecutionException则抛出异常，其他情况则将结果返回。

25.1.3　DRPC Topology端接口及其实现

这部分接口对应于Storm的Topology。Topology的DRPC Spout节点从DRPC服务器获取请求并进行处理。相关的代码如下：

```
1 DistributedRPCInvocations$Iface
2     (^void result [this ^String id ^String result]
3         (let [^Semaphore sem (@id->sem id)]
4            (log-debug "Received result " result " for " id " at " (System/currentTimeMillis))
5            (when sem
6                (swap! id->result assoc id result)
7                (.release sem)
8                )))
9     (^void failRequest [this ^String id]
10        (let [^Semaphore sem (@id->sem id)]
11           (when sem
12               (swap! id->result assoc id (DRPCExecutionException. "Request failed"))
13               (.release sem)
14               )))
15    (^DRPCRequest fetchRequest [this ^String func]
16        (let [^ConcurrentLinkedQueue queue (acquire-queue request-queues func)
17                 ret (.poll queue)]
18           (if ret
19               (do (log-debug "Fetched request for " func " at " (System/currentTimeMillis))
20                   ret)
21               (DRPCRequest. "" ""))
22           ))
```

- 第15~22行实现fetchRequst方法，该方法将被DRPC Spout调用。其输入参数为func，代表函数名字。

- 第16行获得与该函数相对应的请求队列，第17行返回一个请求ret（注意poll操作是阻塞的）。如果ret不为空，那么返回ret，否则构建一个空的DRPCRequest。该请求并不会被真正执行。
- 第2~8行实现result方法，该方法被Topology中某个Bolt调用，用于返回某一请求的结果。第6~7行将结果存储到id->result中，同时释放信号量通知结果将要到来。若不存在对应的sem信号量，则表示该请求已经由于超时或者其他原因而结束了。
- 第9~14行实现fail方法。若查询失败，则将DRPCExecutionException放入结果中，并释放信号量。同样地，该方法也会被Topology中的某个Bolt调用。

25.1.4 启动DRPC服务器

DRPC服务器是基于Thrift的TServer来实现的，所以代码较为简单：

```
1 (defn launch-server!
2    ([]
3       (let [conf (read-storm-config)
4               worker-threads (int (conf DRPC-WORKER-THREADS))
5               queue-size (int (conf DRPC-QUEUE-SIZE))
6               service-handler (service-handler)
7               ;; requests and returns need to be on separate thread pools, since calls to
8               ;; "execute" don't unblock until other thrift methods are called. So if
9               ;; 64 threads are calling execute, the server won't accept the result
10              ;; invocations that will unblock those threads
11              handler-server (THsHaServer. (-> (TNonblockingServerSocket. (int (conf
                   DRPC-PORT)))
12                                             (THsHaServer$Args.)
13                                             (.workerThreads 64)
14                                             (.executorService (ThreadPoolExecutor.
                                                   worker-threads worker-threads
15                                                                60 TimeUnit/SECONDS (Array
                                                                   BlockingQueue.
                                                                   queue-size)))
16                                             (.protocolFactory (TBinaryProtocol$Factory.))
17                                             (.processor (DistributedRPC$Processor.
                                                    service-handler))
18                                             ))
19              invoke-server (THsHaServer. (-> (TNonblockingServerSocket. (int (conf
                    DRPC-INVOCATIONS-PORT)))
20                                             (THsHaServer$Args.)
21                                             (.workerThreads 64)
22                                             (.protocolFactory (TBinaryProtocol$Factory.))
23                                             (.processor (DistributedRPCInvocations
                                                    $Processor. service-handler))
24                                             ))]
25
26              (.addShutdownHook (Runtime/getRuntime) (Thread. (fn [] (.stop handler-
                     server) (.stop invoke-server))))
27               (log-message "Starting Distributed RPC servers...")
28               (future (.serve invoke-server))
29               (.serve handler-server))))
```

- 第4行的worker-threads为线程池中线程的数目，由于目前的execute方法采用了阻塞方式，即只有当获得查询结果时才返回，这就导致目前最多只能有worker-threads个请求被同时执行。
- 第11行实例化handler-server，用于接收用户的请求，它是基于THsHaServer的。第17行传入DistributedRPC$Processor的实现来处理请求。DRPC-PORT定义hanlder-server来收听的端口号。
- 第19行实例化invoke-server，它也是基于THsHaServer的，传入DistributedRPCInvocations$IFace的实现来处理请求。DRPC-INVOCATIONS-PORT定义了invoke-server收听的端口号。
- 第28~29行启动invoke-server和handler-server。这里使用future来启动一个线程，并执行一系列的表达式，执行完成时线程会被回收。这行代码返回了一个future类型的对象以方便之后的引用。之所以使用future方法，是为了以非阻塞的方式调用invoke-server的server方法，然后以阻塞的方式调用handler-server的server方法。

关于 Thrift 服务器的更多信息，请参考 Thrift 的相关文档。

25.2　DRPC 的客户端

Storm中提供了两个DRPC的客户端类，主要是对Thrift产生的客户端进行包装以方便之后的使用。

- DRPCInvocationsClient：用于访问invoke-server。
- DRPCClient：用于访问handler-server。

下面简要分析DRPCClient的主要代码，具体如下：

```
1  public class DRPCClient implements DistributedRPC.Iface {
2      private TTransport conn;
3      private DistributedRPC.Client client;
4      private String host;
5      private int port;
6      private Integer timeout;
7
8      public DRPCClient(String host, int port, Integer timeout) {
9          try {
10             this.host = host;
11             this.port = port;
12             this.timeout = timeout;
13             connect();
14         } catch(TException e) {
15             throw new RuntimeException(e);
16         }
17     }
18
19     private void connect() throws TException {
20         TSocket socket = new TSocket(host, port);
21         if(timeout!=null) {
```

```
22              socket.setTimeout(timeout);
23          }
24          conn = new TFramedTransport(socket);
25          client = new DistributedRPC.Client(new TBinaryProtocol(conn));
26          conn.open();
27      }
28
29      public String execute(String func, String args) throws TException, DRPCExecutionException {
30          try {
31              if(client==null) connect();
32              return client.execute(func, args);
33          } catch(TException e) {
34              client = null;
35              throw e;
36          } catch(DRPCExecutionException e) {
37              client = null;
38              throw e;
39          }
40      }
41  }
```

- 第3行定义DistributedRPC.Client的对象client。该类是由Thrift定义文件产生的。
- 第19~27行定义connect方法。首先定义TSocket对象并构建DistributedRPC.Client对象，然后调用TSocket的open方法来建立连接。
- 第29~39行调用client的execute方法，参数为函数名称func，以及函数的参数args。当有异常发生时，将对client对象进行重置，客户端代码可以捕获异常并完成这种重试。

可以看出，该类是对 Thrift 产生的客户端代码的包装。

25.3 DRPC 中 Spout 节点

DRPC Spout节点从DRPC服务器获得请求，并将请求以消息的形式发送到Topology。在Trident中，DRPC的处理逻辑被编译成为Topology的一部分。DRPC的请求最终被转换成为对一个存储对象的查询。

DRPC Spout需要对发送出去的消息进行跟踪。类DRPCMessageId为其定义MessageId的结构，其中成员变量id为请求序号，index为DRPCClient的序列号。根据index可以得到这个请求是从哪个DRPC服务器获得的，并最终将结果发送至该服务器。集群中可以存在多个DRPC服务器，用户的请求会被发送到其中的一个服务器上，DRPC从该服务器获得请求并进行处理。DRPC服务器之间不存在重复或备份关系。因此，DRPC的消息中需要含有消息id和DRPC服务器序号。类DRPCMessageId的定义如下：

```
private static class DRPCMessageId {
    String id;
    int index;

    public DRPCMessageId(String id, int index) {
```

```
        this.id = id;
        this.index = index;
    }
}
```

接下来分析DRPC Spout的实现，相关代码如下：

```
public class DRPCSpout extends BaseRichSpout {
    public static Logger LOG = LoggerFactory.getLogger(DRPCSpout.class);

    SpoutOutputCollector _collector;
    List<DRPCInvocationsClient> _clients = new ArrayList<DRPCInvocationsClient>();
    String _function;
    String _local_drpc_id = null;

    public DRPCSpout(String function) {
        _function = function;
    }

    public DRPCSpout(String function, ILocalDRPC drpc) {
        _function = function;
        _local_drpc_id = drpc.getServiceId();
    }
```

- _collector为基本的SpoutOutputCollector类型，用于发送消息。
- _clients为DRPC服务器的客户端，为DRPCInvocationsClient类型。由于DRPC服务器和DRPC Spout都有可能存在多个实例，所以同一个Spout可能需要与多个DRPC的服务器进行连接。
- _function为该DRPC Spout要处理的函数。一种类型的DRPC Spout只能处理一种类型的函数。
- _local_drpc_id用来标识是否为模拟模式，为空则表示为非模拟模式。模拟模式与Topology的模拟运行相关。
- 构造函数的参数主要为函数的名称。

DRPC Spout的open函数实现了一个简单的负载均衡算法，其代码如下：

```
1 public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
2     _collector = collector;
3     if(_local_drpc_id==null) {
4         int numTasks = context.getComponentTasks(context.getThisComponentId()).size();
5         int index = context.getThisTaskIndex();
6
7         int port = Utils.getInt(conf.get(Config.DRPC_INVOCATIONS_PORT));
8         List<String> servers = (List<String>) conf.get(Config.DRPC_SERVERS);
9         if(servers == null || servers.isEmpty()) {
10        throw new RuntimeException("No DRPC servers configured for topology");
11        }
12        if(numTasks < servers.size()) {
13            for(String s: servers) {
14                _clients.add(new DRPCInvocationsClient(s, port));
```

```
15              }
16          } else {
17              int i = index % servers.size();
18              _clients.add(new DRPCInvocationsClient(servers.get(i), port));
19          }
20      }
21  }
```

- 第3行表示open函数主要处理非模拟模式，它从DRPC_SERVERS配置项获取DRPC服务器的地址。
- 第12~15行，如果Spout的并行度比DRPC服务器的数目少，则每一个DRPC Spout都会含有一个到DRPC服务器的连接。
- 第17~19行，如果Spout的并行度大于DRPC的服务器数目，则只连接特定的DRPC服务器。

目前的这个算法还是比较简单的。

DRPC Spout的输出含有两列：args表示函数的参数，return-info的模式为<requestId，host，port>，用来标识该请求是从哪个DRPC服务器获得的：

```
@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("args", "return-info"));
}
```

下面来看DRPC Spout中nextTuple函数的实现：

```
1 public void nextTuple() {
2     boolean gotRequest = false;
3     if(_local_drpc_id==null) {
4         for(int i=0; i<_clients.size(); i++) {
5             DRPCInvocationsClient client = _clients.get(i);
6             try {
7                 DRPCRequest req = client.fetchRequest(_function);
8                 if(req.get_request_id().length() > 0) {
9                     Map returnInfo = new HashMap();
10                    returnInfo.put("id", req.get_request_id());
11                    returnInfo.put("host", client.getHost());
12                    returnInfo.put("port", client.getPort());
13                    gotRequest = true;
14                    _collector.emit(new Values(req.get_func_args(), JSONValue.toJSONString
                          (returnInfo)), new DRPCMessageId(req.get_request_id(), i));
15                    break;
16                }
17            } catch (TException e) {
18                LOG.error("Failed to fetch DRPC result from DRPC server", e);
19            }
20        }
21    }
22    if(!gotRequest) {
23        Utils.sleep(1);
24    }
25 }
```

- 第4~19行依次遍历DRPC服务器的客户端并调用fetchRequst方法。如果返回了非空的请求，则构建一条消息并发送出去，消息的内容为请求函数的参数和序列化后的请求源的信息。它使用通过请求序号和服务器索引构建的DRPCMessageId对象来跟踪消息。
- 请求序号会起到事务序号的作用。在Trident中，该序号被放在了默认的第1列，因此这里没有办法再去放置消息源的信息。目前，Trident采用连接（join）的方式来解决这个问题：即产生最终结果的Bolt节点将同时接收DRPC Spout发来的消息和从其他Bolt节点发送来的结果消息，然后按照请求序号进行连接，以获得请求的来源信息，并最终将结果发送到DRPC服务器。
- 由于请求序号起到了事务序号的作用，产生结果的Bolt只有在收到DRPC Spout发送的消息以及从其他Bolt发送过来的结果之后，finishBatch方法才会被调用。

接下来，看一下fail方法的实现代码：

```
public void fail(Object msgId) {
    DRPCMessageId did = (DRPCMessageId) msgId;
    DistributedRPCInvocations.Iface client;

    if(_local_drpc_id == null) {
        client = _clients.get(did.index);
    } else {
        client = (DistributedRPCInvocations.Iface) ServiceRegistry.getService(_local_drpc_id);
    }
    try {
        client.failRequest(did.id);
    } catch (TException e) {
        LOG.error("Failed to fail request", e);
    }
}
```

当消息处理失败后，DRPC Spout的fail方法将被调用，它根据msgId找到请求的来源，然后通过client的failRequest方法通知DRPC服务器。

目前，ack方法不需要完成任何任务。当ack方法被回调时，DRPC 服务器已经收到了从Bolt发送的结果。将来可以在此处增加一些清理操作等。

```
public void ack(Object msgId) {
}
```

DRPC Spout是Trident中几个基本Spout类型之一。

25.4　DRPC Spout 的执行器

类RichSpoutBatchTriggerer在一定程度上完成了类似于TridentBoltExecutor的功能。它发送协调消息，并利用这些消息来判断属于同一个事务的消息是否处理结束，目前主要用于封装DRPC Spout。该类的代码如下：

```
public class RichSpoutBatchTriggerer implements IRichSpout {

    String _stream;
    IRichSpout _delegate;
    List<Integer> _outputTasks;
    Random _rand;
    String _coordStream;
    Map<Long, Long> _msgIdToBatchId = new HashMap();
    Map<Long, FinishCondition> _finishConditions = new HashMap();
    public RichSpoutBatchTriggerer(IRichSpout delegate, String streamName, String batchGroup) {
        _delegate = delegate;
        _stream = streamName;
        _coordStream = TridentBoltExecutor.COORD_STREAM(batchGroup);
    }
}
```

- _stream为通常的消息流。
- _coordStream用于发送协调消息，该流的前缀为“$coord-”，这与TridentBoltExecutor是相同的。
- _delegate是被封装的Spout。
- _outputTasks为接收该Spout消息的Bolt节点。
- _rand为一个随机数产生器，产生的随机数用于进行消息跟踪。
- _msgIdToBatchId为从消息跟踪序号到事务序号的映射关系。在DRPC Spout中，消息跟踪序号为请求序号。同一条消息可能被多个Task接收，而每个接收端都会对应于一个新的事务序号。
- _finishContions主要用来表示一条消息是否已经处理完成，它的键为MessageId。FinishCondition包含_delete发送消息携带的MessageId和BatchId的集合。

类StreamOverrideCollector对基本的SpoutOutputCollect进行了封装，其代码如下：

```
1 class StreamOverrideCollector implements ISpoutOutputCollector {
2
3     SpoutOutputCollector _collector;
4
5     public StreamOverrideCollector(SpoutOutputCollector collector) {
6         _collector = collector;
7     }
8
9     @Override
10    public List<Integer> emit(String ignore, List<Object> values, Object msgId) {
11        long batchIdVal = _rand.nextLong();
12        Object batchId = new RichSpoutBatchId(batchIdVal);
13        FinishCondition finish = new FinishCondition();
14        finish.msgId = msgId;
15        List<Integer> tasks = _collector.emit(_stream, new ConsList(batchId, values));
16        Set<Integer> outTasksSet = new HashSet<Integer>(tasks);
17        for(Integer t: _outputTasks) {
18            int count = 0;
19            if(outTasksSet.contains(t)) {
20                count = 1;
```

25

```
21            }
22            long r = _rand.nextLong();
23            _collector.emitDirect(t, _coordStream, new Values(batchId, count), r);
24            finish.vals.add(r);
25        }
26        _finishConditions.put(batchIdVal, finish);
27        return tasks;
28    }
29
30    @Override
31    public void emitDirect(int task, String ignore, List<Object> values, Object msgId) {
32        throw new RuntimeException("Trident does not support direct emits from spouts");
33    }
34
35    @Override
36    public void reportError(Throwable t) {
37        _collector.reportError(t);
38    }
39 }
```

- ❑ 第10~29行实现的emit方法，是这个类的核心。RichSpoutBatchId实现了IBatchId接口，但其attempId始终为0。事务尝试也实现了IBatchId接口，它则主要用于事务Topology。
- ❑ 第10~11行用随机数的方式产生了一个事务尝试序号。
- ❑ 第12~13行定义了一个FinishCondition变量，并将输入消息的MessageId赋值给它。
- ❑ 第15行将消息发送出去，消息的第1列为通过随机方法产生的事务序号。可以看到此处并没有对消息进行跟踪，由被代理的Spout发送出来的消息的MessageId只是被存储于FinishCondition变量中。
- ❑ 第15行返回的TaskId为_outputTasks的子集。
- ❑ 第17~25行通过emitDirect的方式向其协调流发送一条消息，格式为<事务序号, 消息条数（0或1）>，同时产生一个随机数用来跟踪消息。

目前的代码中有一个Bug，需要在第25行之后添加下面的代码来构建正确的映射关系：

```
_msgIdToBatchId.put(r, batchIdVal);
```

第26行将存储batchId和FinishCondition间的关系。最后来看流的声明方法，相关代码如下：

```
public void declareOutputFields(OutputFieldsDeclarer declarer) {
    Fields outFields = TridentUtils.getSingleOutputStreamFields(_delegate);
    outFields = TridentUtils.fieldsConcat(new Fields("$id$"), outFields);
    declarer.declareStream(_stream, outFields);
    // try to find a way to merge this code with what's already done in TridentBoltExecutor
    declarer.declareStream(_coordStream, true, new Fields("id", "count"));
}
```

RichSpoutBatchTriggerer的declareOutputFields函数声明了两个流。

- ❑ _stream：其列为“id”加上原有的列。
- ❑ _coordStream：其列为“id”和消息数目。

由于只对发送到协调流的消息进行了跟踪，因此当收到所有发送到协调流的消息后，将调

用代理类的ack方法，该方法的代码如下：

```
public void ack(Object msgId) {
    Long batchId = _msgIdToBatchId.remove((Long) msgId);
    FinishCondition cond = _finishConditions.get(batchId);
    if(cond!=null) {
        cond.vals.remove((Long) msgId);
        if(cond.vals.isEmpty()) {
            _finishConditions.remove(batchId);
            _delegate.ack(cond.msgId);
        }
    }
}
```

fail方法的实现与ack方法类似，只不过其在收到任何一条失败消息时均会直接调用_delegate的fail方法。fail方法的代码如下：

```
public void fail(Object msgId) {
    Long batchId = _msgIdToBatchId.remove((Long) msgId);
    FinishCondition cond = _finishConditions.remove(batchId);
    if(cond!=null) {
        _delegate.fail(cond.msgId);
    }
}
```

这里需要根据msgId找到batchId。msgId为发送协调消息时产生的随机数。

其他的方法都较为直观，这里不再一一讨论。

25.5 completeDRPC 操作

在Trident中，需要添加DRPC Spout节点，而当DRPC请求被处理后，又需要有节点可以将结果发送回DRPC 服务器。类TridentTopology的completeDRPC操作即可用来完成对这部分节点的构建，该类的定义如下：

```
1 private static void completeDRPC(DefaultDirectedGraph<Node, IndexedEdge> graph, Map<String,
     List<Node>> colocate, UniqueIdGen gen) {
2    List<Set<Node>> connectedComponents = new ConnectivityInspector<Node,
         IndexedEdge>(graph).connectedSets();
3
4    for(Set<Node> g: connectedComponents) {
5       checkValidJoins(g);
6    }
7
8    TridentTopology helper = new TridentTopology(graph, colocate, gen);
9    for(Set<Node> g: connectedComponents) {
10       SpoutNode drpcNode = getDRPCSpoutNode(g);
11       if(drpcNode!=null) {
12          Stream lastStream = new Stream(helper, null, getLastAddedNode(g));
13          Stream s = new Stream(helper, null, drpcNode);
14          helper.multiReduce(
15              s.project(new Fields("return-info"))
```

```
16                .batchGlobal(),
17               lastStream.batchGlobal(),
18               new ReturnResultsReducer(),
19               new Fields());
20         }
21     }
22 }
```

- 第2行根据有向图的最大连通子图算法获得最大连通子图集合。第6~7行对每一个连通集合进行检查：任何一个连通集合中，不能同时含有基本类型和DRPC类型两类Spout节点。当然，一个连通子图中是可以含有多个基本类型Spout节点的。
- 第9~21行对含有DRPC Spout节点连通子图进行处理，此处需要对两个流进行连接操作。
- 第13行以DRPC Spout节点为输入创建一个新的流s；第15行调用该流的映射操作，创建一个处理节点，该节点的输出中将含有一列return-info信息。
- 第12行通过getLastAddedNode方法获得该连通子图中最后添加的节点，即creationIndex值最大的节点。在DRPC对应的连通子图中，该节点是查询消息结果所对应的节点。
- 同时，调用这两个流的batchGlobal操作来创建分区节点，使得具有相同事务尝试序号的节点可以到达相同的目标节点。
- 第4~19行利用multiReduce操作对这两个流进行连接处理。

类ReturnResultsReducer用于处理并返回结果到DRPC Server。

由于这两个流的源头为同一个DRPC Spout节点，具有相同的事务序号，因此multiReduce节点实际上是根据事务序号执行的连接操作。

completeDRPC操作构建的节点如图25-2所示。

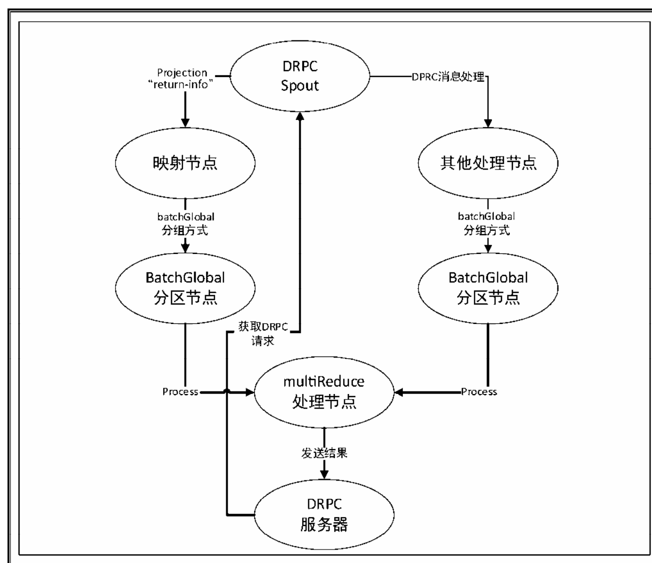


图25-2　completeDRPC操作

25.6 返回 DRPC 结果

ReturnResultsReducer类用来将DRPC的结果返回给DRPC服务器。

ReturnResultsState类用来存储DPRC的结果，是ReturnResultsReducer类操作的数据。ReturnResultsReducer操作所在的Bolt节点将接收两个流，returnInfo从其中一个流获得，包含了这次DRPC请求的来源DRPC服务器信息，请求结果需要发送给该DRPC服务器。而results为消息列表，从另外两个流获得，代表了查询的结果。该类的定义如下：

```
public static class ReturnResultsState {
    List<TridentTuple> results = new ArrayList<TridentTuple>();
    String returnInfo;

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this);
    }
}
```

下面我们再来看ReturnResultsReducer类的实现：

```
1  public class ReturnResultsReducer implements MultiReducer<ReturnResultsState> {
2     boolean local;
3
4     Map<List, DRPCInvocationsClient> _clients = new HashMap<List, DRPCInvocationsClient>();
5
6
7     @Override
8     public void prepare(Map conf, TridentMultiReducerContext context) {
9         local = conf.get(Config.STORM_CLUSTER_MODE).equals("local");
10    }
11
12    @Override
13    public ReturnResultsState init(TridentCollector collector) {
14        return new ReturnResultsState();
15    }
16
17    @Override
18    public void execute(ReturnResultsState state, int streamIndex, TridentTuple input,
         TridentCollector collector) {
19        if(streamIndex==0) {
20            state.returnInfo = input.getString(0);
21        } else {
22            state.results.add(input);
23        }
24    }
25
26    @Override
27    public void complete(ReturnResultsState state, TridentCollector collector) {
28        // only one of the multireducers will receive the tuples
29        if(state.returnInfo!=null) {
```

```
30            String result = JSONValue.toJSONString(state.results);
31            Map retMap = (Map) JSONValue.parse(state.returnInfo);
32            final String host = (String) retMap.get("host");
33            final int port = Utils.getInt(retMap.get("port"));
34            String id = (String) retMap.get("id");
35            DistributedRPCInvocations.Iface client;
36            if(local) {
37                client = (DistributedRPCInvocations.Iface) ServiceRegistry.getService(host);
38            } else {
39                List server = new ArrayList() {{
40                    add(host);
41                    add(port);
42                }};
43
44                if(!_clients.containsKey(server)) {
45                    _clients.put(server, new DRPCInvocationsClient(host, port));
46                }
47                client = _clients.get(server);
48            }
49
50            try {
51                client.result(id, result);
52            } catch(TException e) {
53                collector.reportError(e);
54            }
55        }
56    }
57
58    @Override
59    public void cleanup() {
60        for(DRPCInvocationsClient c: _clients.values()) {
61            c.close();
62        }
63    }
64 }
```

- 第4行定义的_clients成员变量用来缓存客户端到DRPC服务器。
- 第13~15行的init方法用于初始化ReturnResultsState对象。
- 第18~24行的execute方法会根据输入消息的流序号来判断信息属于哪种类型，或者为return-info或者为处理结果。
- 第27~56行的complete方法用于表明一个请求已经处理结束。若包含处理结果，则从return-info中获取DRPC服务器的主机名和端口号。
- 第37行获得DRPC服务器的客户端client，第51行调用result方法将结果返回。第59~63行中的close方法则会将DRPC客户端关闭。

第 26 章

Trident的Topology构建器

26

操作流、执行Topology以及运转DRPC的过程使Trident获得了基本的Spout和Bolt节点。类TridentTopologyBuilder将构建Storm所运行的Topology，并负责生命系统流以及Topology与该流的收听关系，这部分是较为复杂的。

26.1 基本工具函数

本节对构建Topology的过程中需要用到的一些重要函数进行介绍，它们是理解的难点。

26.1.1 committerBatches

当一个处理节点中含有State对象时，其committer变量将被设置为真。若一个节点组中含有一个具有存储对象的节点，则该节点组将被返回。committerBatches函数用于返回所有满足条件的节点组，其代码如下：

```
private static Set<String> committerBatches(Group g, Map<Node, String> batchGroupMap) {
    Set<String> ret = new HashSet();
    for(Node n: g.nodes) {
       if(n instanceof ProcessorNode) {
           if(((ProcessorNode) n).committer) {
               ret.add(batchGroupMap.get(n));
           }
       }
    }
    return ret;
}
```

26.1.2 fleshOutStreamBatchIds

该函数返回从全局流序号（GlobalStreamId）到节点组的映射关系，它主要用于构建系统的全局流序号与节点组的映射关系。

变量_batchIds中含有用户定义的从Topology全局流序号到节点组的映射关系，fleshOutputStreamBatchIds函数将这一映射关系经过计算然后添加进_batchlds中去。具体涉及的流为：

- MasterBatchCoordinator节点的$commit和$batch流；
- _spouts中的$batch流；
- _bolt中的$coord-bgx流，bgx代表节点组所对应的序号。

其中，includeCommitStream变量表示是否包含MasterBatchCoordinator的$commit流。注意并不是每一个节点组都存在MasterBatchCoordinator节点。fleshOutStreamBatchIds函数的代码如下：

```
Map<GlobalStreamId, String> fleshOutStreamBatchIds(boolean includeCommitStream) {
    Map<GlobalStreamId, String> ret = new HashMap<GlobalStreamId, String>(_batchIds);
    Set<String> allBatches = new HashSet(_batchIds.values());
    for(String b: allBatches) {
        ret.put(new GlobalStreamId(masterCoordinator(b), MasterBatchCoordinator.BATCH_STREAM_ID), b);
        if(includeCommitStream) {
            ret.put(new GlobalStreamId(masterCoordinator(b), MasterBatchCoordinator.
                COMMIT_STREAM_ID), b);
        }
        // DO NOT include the success stream as part of the batch. it should not trigger coordination
            tuples,
        // and is just a metadata tuple to assist in cleanup, should not trigger batch tracking
    }

    for(String id: _spouts.keySet()) {
        TransactionalSpoutComponent c = _spouts.get(id);
        if(c.batchGroupId!=null) {
            ret.put(new GlobalStreamId(spoutCoordinator(id), MasterBatchCoordinator.BATCH_
                STREAM_ID), c.batchGroupId);
        }
    }

    //this takes care of setting up coord streams for spouts and bolts
    for(GlobalStreamId s: _batchIds.keySet()) {
        String b = _batchIds.get(s);
        ret.put(new GlobalStreamId(s.get_componentId(), TridentBoltExecutor.COORD_STREAM(b)), b);
    }

    return ret;
}
```

26.1.3 getOutputStreamBatchGroups

该方法返回一个输出流所对应的节点组，其代码如下：

```
private static Map<String, String> getOutputStreamBatchGroups(Group g, Map<Node, String> batchGroupMap) {
    Map<String, String> ret = new HashMap();
    Set<PartitionNode> externalGroupOutputs = externalGroupOutputs(g);
    for(PartitionNode n: externalGroupOutputs) {
        ret.put(n.streamId, batchGroupMap.get(n));
    }
    return ret;
}
```

26.2 TridentTopologyBuilder

本节将讨论TridentTopologyBuilder的类实现，主要分析其是如何构建Spout和Bolt节点的，并着重讲述其中有关于系统流设置的内容。下面首先对该类的数据进行分析。

26.2.1 成员变量

TridentTopologyBuilder的数据成员分析如下：

```
public class TridentTopologyBuilder {
    Map<GlobalStreamId, String> _batchIds = new HashMap();
    Map<String, TransactionalSpoutComponent> _spouts = new HashMap();
    Map<String, SpoutComponent> _batchPerTupleSpouts = new HashMap();
    Map<String, Component> _bolts = new HashMap();
}
```

- _batchIds中存储了Topology中每一个从全局流序号到节点组的映射关系。
- _spouts存储了从SpoutId到事务组件的映射关系。事务组件定义如下：

```
private static class SpoutComponent {
    public Object spout;
    public Integer parallelism;
    public List<Map> componentConfs = new ArrayList<Map>();
    String batchGroupId;
    String streamName;
}
private static class TransactionalSpoutComponent extends SpoutComponent {
    public String commitStateId;
}
```

- batchGroupId为Spout节点所对应的节点组。
- commitStateId是事务组件特有的，对应于ZooKeeper中路径，用于存储元数据。
- _batchPerTupleSpouts 用来存储从SpoutId到SpoutComponent的映射关系。batchPerTupleSpouts中存储的Spout每个事务只发送一条消息（如DRPC类型的Spout节点）。
- _bolts保存了系统中所有的Bolt节点，其中Component的定义如下：

```
private static class Component {
    public ITridentBatchBolt bolt;
    public Integer parallelism;
    public List<InputDeclaration> declarations = new ArrayList<InputDeclaration>();
    public List<Map> componentConfs = new ArrayList<Map>();
    public Set<String> committerBatches;
}
```

committerBatches成员变量用于表明哪些节点组与该节点有联系。该Bolt节点将收听这些节点组所对应的MasterCoordinator节点的$commit流。

26.2.2　设置Spout节点

本小节讨论Trident是如何设置Spout节点并构建系统流的，相关代码如下：

```
1 TopologyBuilder builder = new TopologyBuilder();
2 Map<GlobalStreamId, String> batchIdsForSpouts = fleshOutStreamBatchIds(false);
3 Map<GlobalStreamId, String> batchIdsForBolts = fleshOutStreamBatchIds(true);
4
5 Map<String, List<String>> batchesToCommitIds = new HashMap<String, List<String>>();
6 Map<String, List<ITridentSpout>> batchesToSpouts = new HashMap<String, List
     <ITridentSpout>>();
7
8 for(String id: _spouts.keySet()) {
9     TransactionalSpoutComponent c = _spouts.get(id);
10    if(c.spout instanceof IRichSpout) {
11
12        //TODO: wrap this to set the stream name
13        builder.setSpout(id, (IRichSpout) c.spout, c.parallelism);
14    } else {
15        String batchGroup = c.batchGroupId;
16        if(!batchesToCommitIds.containsKey(batchGroup)) {
17            batchesToCommitIds.put(batchGroup, new ArrayList<String>());
18        }
19        batchesToCommitIds.get(batchGroup).add(c.commitStateId);
20
21        if(!batchesToSpouts.containsKey(batchGroup)) {
22            batchesToSpouts.put(batchGroup, new ArrayList<ITridentSpout>());
23      }
24        batchesToSpouts.get(batchGroup).add((ITridentSpout) c.spout);
25
26
27        BoltDeclarer scd =
28           builder.setBolt(spoutCoordinator(id), new TridentSpoutCoordinator
                (c.commitStateId, (ITridentSpout) c.spout))
29           .globalGrouping(masterCoordinator(c.batchGroupId), MasterBatchCoordinator.
                BATCH_STREAM_ID)
30           .globalGrouping(masterCoordinator(c.batchGroupId), MasterBatchCoordinator.
                SUCCESS_STREAM_ID);
31
32        for(Map m: c.componentConfs) {
33            scd.addConfigurations(m);
34        }
35
36        Map<String, TridentBoltExecutor.CoordSpec> specs = new HashMap();
37        specs.put(c.batchGroupId, new CoordSpec());
38        BoltDeclarer bd = builder.setBolt(id,
39            new TridentBoltExecutor(
40                new TridentSpoutExecutor(
41                    c.commitStateId,
42                    c.streamName,
43                    ((ITridentSpout) c.spout)),
44                    batchIdsForSpouts,
45            specs),
```

```
46            c.parallelism);
47        bd.allGrouping(spoutCoordinator(id), MasterBatchCoordinator.BATCH_STREAM_ID);
48        bd.allGrouping(masterCoordinator(batchGroup), MasterBatchCoordinator.
              SUCCESS_STREAM_ID);
49        if(c.spout instanceof ICommitterTridentSpout) {
50            bd.allGrouping(masterCoordinator(batchGroup), MasterBatchCoordinator.
                  COMMIT_STREAM_ID);
51        }
52        for(Map m: c.componentConfs) {
53            scd.addConfigurations(m);
54        }
55    }
56 }
57
58 for(String id: _batchPerTupleSpouts.keySet()) {
59     SpoutComponent c = _batchPerTupleSpouts.get(id);
60         SpoutDeclarer d = builder.setSpout(id, new RichSpoutBatchTriggerer
               ((IRichSpout) c.spout, c.streamName, c.batchGroupId), c.parallelism);
61
62     for(Map conf: c.componentConfs) {
63         d.addConfigurations(conf);
64     }
65 }
66
67 for(String batch: batchesToCommitIds.keySet()) {
68     List<String> commitIds = batchesToCommitIds.get(batch);
69     builder.setSpout(masterCoordinator(batch), new MasterBatchCoordinator(commitIds,
          batchesToSpouts.get(batch)));
70 }
```

- 第58~65行设置_batchPerTupleSpout中的Spout节点，这里利用了RichSpoutBatchTriggerer类对其进行包装。
- 第5行的batchesToCommitIds变量用来存储从节点组序号到ITridentSpout中commitStateId的映射关系。
- 第6行的batchesToSpouts变量用来存储从节点组序号到Spout的映射关系，一个BatchGroup中会含有多个ITridentSpout节点。
- 第67~70行设置Trident中的Spout节点，利用类MasterBatchCoordinator对节点组中所有ITridentSpout进行封装。可以看出若节点组中没有ITridentSpout节点，则也就没有MasterBatchCoordinator节点。MasterBatchCoordinator节点会向$commit、$batch和$success流发送消息。
- 第8~56行处理每一个_spouts节点。第10~13行，若节点为IRichSpout类型，则直接调用setSpout方法进行设置。第15~24行更新batchesToCommitIds和batchesToSpouts变量，为设置MasterBatchCoordinator节点做准备。
- 第27~30行设置了一个Bolt节点，并利用TridentSpoutCoordinator对Spout进行封装，该Bolt节点会收听MasterBatchCoordinator节点所对应的$batch和$success流，收听方式为全局分组方式。

- 第38~46行又设置了一个Bolt节点，并利用TridentBoltExecutor和TridentSpoutExecutor对Spout进行封装。第47~51行表示该节点将收听MasterBatchCoordinator节点的$success流和TridentSpoutCoordinator的$batch流。若Spout实现了ICommitterTridentSpout接口，则还需要收听MasterBatchCoordinator节点的$commit流。

即用户定义的ITridentSpout将被部署到3个节点上运行，如图26-1所示。读者可结合代码自行分析。

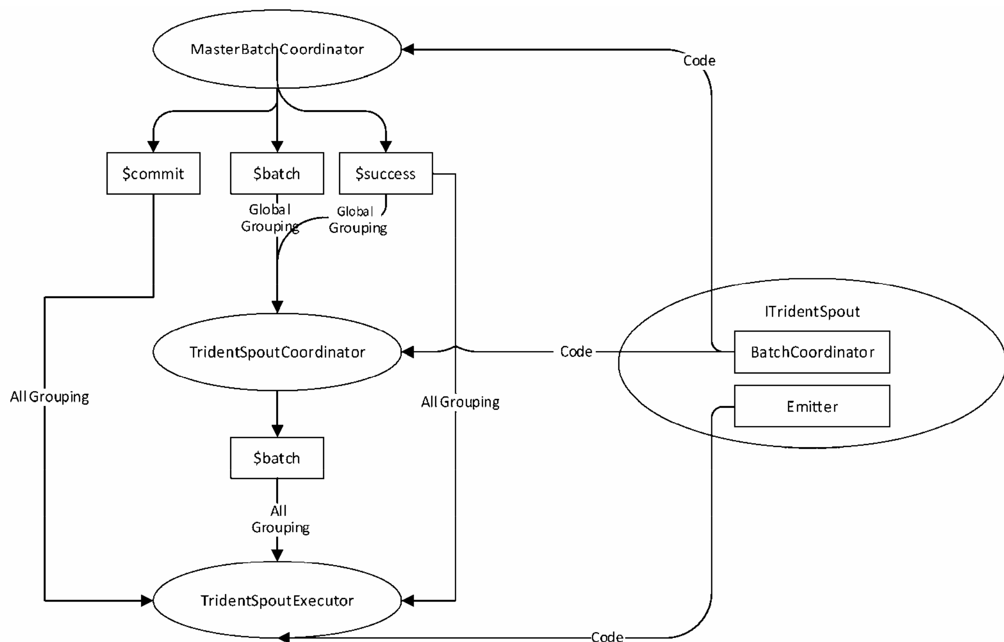


图26-1　ITridentSpout的部署以及流

26.2.3　设置Bolt节点

本节讨论Trident是如何设置Bolt节点并构建系统流，相关代码如下：

```
1 for(String id: _bolts.keySet()) {
2     Component c = _bolts.get(id);
3
4     Map<String, CoordSpec> specs = new HashMap();
5
6     for(GlobalStreamId s: getBoltSubscriptionStreams(id)) {
7         String batch = batchIdsForBolts.get(s);
8         if(!specs.containsKey(batch)) specs.put(batch, new CoordSpec());
9         CoordSpec spec = specs.get(batch);
10        CoordType ct;
11        if(_batchPerTupleSpouts.containsKey(s.get_componentId())) {
```

```
12            ct = CoordType.single();
13        } else {
14            ct = CoordType.all();
15        }
16        spec.coords.put(s.get_componentId(), ct);
17    }
18
19    for(String b: c.committerBatches) {
20        specs.get(b).commitStream = new GlobalStreamId(masterCoordinator(b),
            MasterBatchCoordinator.COMMIT_STREAM_ID);
21    }
22
23    BoltDeclarer d = builder.setBolt(id, new TridentBoltExecutor(c.bolt,
        batchIdsForBolts, specs), c.parallelism);
24    for(Map conf: c.componentConfs) {
25        d.addConfigurations(conf);
26    }
27
28    for(InputDeclaration inputDecl: c.declarations) {
29       inputDecl.declare(d);
30    }
31
32   Map<String, Set<String>> batchToComponents = getBoltBatchToComponentSubscriptions(id);
33    for(String b: batchToComponents.keySet()) {
34        for(String comp: batchToComponents.get(b)) {
35            d.directGrouping(comp, TridentBoltExecutor.COORD_STREAM(b));
36        }
37    }
38
39    for(String b: c.committerBatches) {
40        d.allGrouping(masterCoordinator(b), MasterBatchCoordinator.COMMIT_STREAM_ID);
41    }
42 }
```

- 第4行的specs变量用来存储每个节点组所对应的协调关系定义（CoordSpec）。协调关系定义用来描述该节点将从哪些节点获取协调消息。
- 第6~17行设置该Bolt的父节点的协调关系，如果父节点属于_batchPerTupleSpouts，则将CoordType设置为single，表示即便该组件存在多个并行度，该Bolt也只能从其中一个父节点处接收一条消息。例如，DRPC Spout中每条Spout所发送的消息都对应于一个新的请求，该请求会发送到下游的某一个Bolt节点上。其他类型父组件的CoordType为all，表示为该节点需要从父节点的所有并行度中收取协调消息。
- 第39~41行表示若该节点中含有State对象，则需要收听MasterCoordinator的$commit流，于是该节点可以获得一个合适的时机去更新State对象。
- 第23行表示Bolt节点都是通过TridentBoltExecutor进行封装的。
- 第28~30行添加用户的流声明。
- 第32~37行该Bolt节点通过直接消息分组的方式收听父节点的$coord-bgx流，bgx为节点组序号。

26.3　一个例子

最后看一下Storm-Starter中Trident Word Count Topology的定义，以及实际运行的Topology。Topology的定义如下：

```
1 TridentTopology topology = new TridentTopology();
2 TridentState wordCounts =
3    topology.newStream("spout1", spout)
4    .parallelismHint(1)
5    .each(new Fields("sentence"), new Split(), new Fields("word"))
6    .groupBy(new Fields("word"))
7    .persistentAggregate(new MemoryMapState.Factory(),
8                           new Count(), new Fields("count"))
9     .parallelismHint(1);
10
11 topology.newDRPCStream("words", drpc).name("DRPCWord")
12    .each(new Fields("args"), new Split(), new Fields("word"))
13    .parallelismHint(5)
14    .groupBy(new Fields("word")).name("DRPCGROUPBY")
15    .stateQuery(wordCounts, new Fields("word"), new MapGet(), new Fields("count"))
16    .each(new Fields("count"), new FilterNull())
17    .aggregate(new Fields("count"), new Sum(), new Fields("sum"))
18    ;
19    return topology.build();
```

首先，由于该例子没有恰当的使用流的name操作，于是Component的名字基本上为系统随机取得的默认名字，这对于调试过程来讲简直是灾难。系统实际运行的Topology如图26-2所示。

- 系统中只有一个Mastercoord-bg0节点，这是因为系统中只存在一个节点组含有State存储节点。
- 用户定义的协调Spout节点被部署到了Mastercoord-bg0和spout0上，Mastercoord-bg0会调用isReady函数以确定是否该发送一个事务消息。spout0收到该事务消息后，则调用initializeTransaction函数去初始化一个事务。
- Trident将Bolt的Split操作和分区内部的Count操作放在了$b-1节点上。
- spout1代表了另外一个节点组的Spout节点，它是DRPC Spout，该Spout从DRPC服务获得一个请求，并向Topology发送消息。
- b-2节点只含有与DRPC请求源相关的消息。例如，该请求从哪个DRPC服务器收到的。
- 节点b-4执行了全局的聚集操作，同时将结果写入State对象。而且，它还需要完成来自另外一个节点组的DRPC 查询操作，并将查询结果发送出去。
- 节点b-5会按照事务序号对查询结果和消息来源进行连接，由此确定查询结果应该发送到哪一个DRPC服务器，然后完成这个发送操作。

图26-2中的虚线表示为协调信息和事务信息，实线为具体的数据信息。

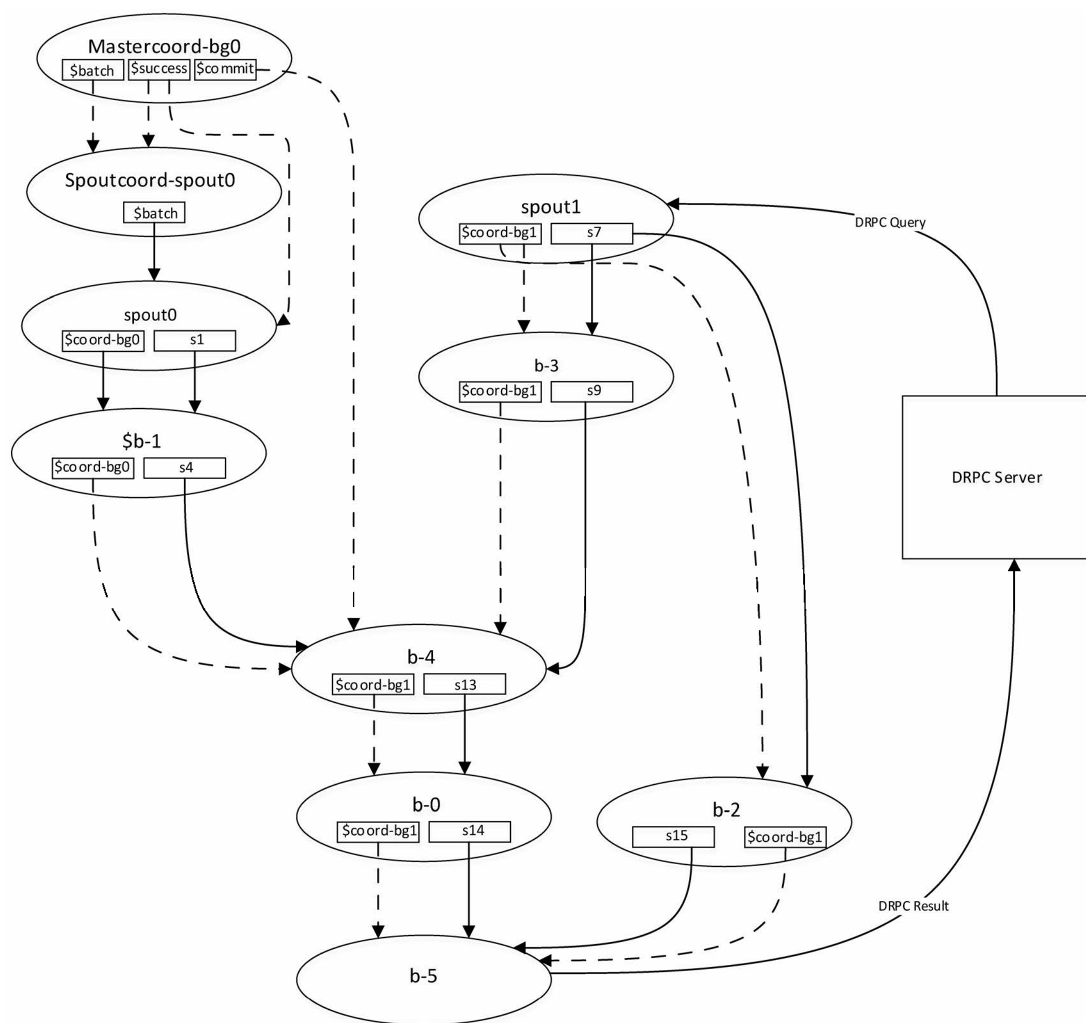


图26-2 Trident Topology实例

第 27 章 多语言

Storm是支持多语言的，它采用标准输入输出来与用其他语言定义的Spout或Bolt进行通信。读者可参考下面链接中的文档来了解Storm的多语言支持和通信协议。

- https://github.com/nathanmarz/storm/wiki/Using-non-JVM-languages-with-Storm
- https://github.com/nathanmarz/storm/wiki/Multilang-protocol

通过标准输入输出进行进程间通信有时候并不能达到性能要求，这时需要用户自己来实现相应的协议从而获得更高的效率。

本章对其中几个重要的实现类进行简要介绍。

27.1 ShellProcess

ShellProcess类用于加载一个由其他语言定义的进程，并通过STDIN和STDOUT进行通信。进程的参数被放在command类成员变量中，该变量将获得子进程的标准输入输出和运行错误，从而完成与子进程间的通信。ShellProcess类的定义如下：

```
1 public class ShellProcess {
2     private DataOutputStream processIn;
3     private BufferedReader processOut;
4     private InputStream processErrorStream;
5     private Process _subprocess;
6     private String[] command;
7
8     public ShellProcess(String[] command) {
9         this.command = command;
10    }
11
12    public Number launch(Map conf, TopologyContext context) throws IOException {
13        ProcessBuilder builder = new ProcessBuilder(command);
14        builder.directory(new File(context.getCodeDir()));
15        _subprocess = builder.start();
16
17        processIn = new DataOutputStream(_subprocess.getOutputStream());
18        processOut = new BufferedReader(new InputStreamReader(_subprocess.getInputStream()));
19        processErrorStream = _subprocess.getErrorStream();
20
21        JSONObject setupInfo = new JSONObject();
```

```
22          setupInfo.put("pidDir", context.getPIDDir());
23          setupInfo.put("conf", conf);
24          setupInfo.put("context", context);
25          writeMessage(setupInfo);
26
27          return (Number)readMessage().get("pid");
28      }
29
30      public void destroy() {
31          _subprocess.destroy();
32      }
33 }
```

- 第12~28行定义的lanuch方法用于启动进程。第21行构建了一个JSONObject类型的setupInfo，里面含有Worker数据目录下的PID目录、Topology的配置项和上下文信息等内容。子进程可以创建一个以自身进程ID为名字的空文件，并放置于PID目录下，这样Worker就可以对其启动的子进程进行管理了。
- 第27行，ShellProcess希望用户返回其进程ID（实际上最好还是由ShellProcess类来创建PID文件，但目前仍需要用户创建PID文件）。

ShellProcess定义了readMessage、writeMessage和getErrorString方法来与子进程进行通信。通信结束的标志为end行。下面以readMessage为例进行代码分析：

```
1 public JSONObject readMessage() throws IOException {
2     String string = readString();
3     JSONObject msg = (JSONObject)JSONValue.parse(string);
4     if (msg != null) {
5         return msg;
6     } else {
7         throw new IOException("unable to parse: " + string);
8     }
9 }
10
11 private String readString() throws IOException {
12    StringBuilder line = new StringBuilder();
13
14    //synchronized (processOut) {
15        while (true) {
16            String subline = processOut.readLine();
17            if(subline==null) {
18                StringBuilder errorMessage = new StringBuilder();
19                errorMessage.append("Pipe to subprocess seems to be broken!");
20                if (line.length() == 0) {
21                    errorMessage.append(" No output read.\n");
22                }
23                else {
24                    errorMessage.append(" Currently read output: " + line.toString() + "\n");
25                   }
26                   errorMessage.append("Shell Process Exception:\n");
27                   errorMessage.append(getErrorsString() + "\n");
28                   throw new RuntimeException(errorMessage.toString());
```

```
29                }
30                if(subline.equals("end")) {
31                    break;
32                }
33                if(line.length()!=0) {
34                    line.append("\n");
35                }
36                line.append(subline);
37            }
38            //}
39
40        return line.toString();
41    }
```

- 第11~41行定义了私有的readString方法。该方法会不断调用子进程的标准输出来获取字符串，如果所读到的当前行为“end”，则认为消息已经结束。
- 第17行，若收回的subline为空，表明子进程已经不存在了，此时会准备错误消息并抛出异常（RuntimeException）。
- 第1~9行定义的readString方法将按照JSON的格式对消息进行解析并返回JSONObject对象。

27.2　ShellBolt

ShellBolt通过自定义的协议与子进程进行通信。为了提高效率，ShellBolt定义了两个线程，一个用于向子进程发送消息，一个用于从子进程接收消息。

27.2.1　成员变量

ShellBolt 对象的数据分析如下：

```
1 public class ShellBolt implements IBolt {
2     public static Logger LOG = LoggerFactory.getLogger(ShellBolt.class);
3     Process _subprocess;
4     OutputCollector _collector;
5     Map<String, Tuple> _inputs = new ConcurrentHashMap<String, Tuple>();
6
7     private String[] _command;
8     private ShellProcess _process;
9     private volatile boolean _running = true;
10    private volatile Throwable _exception;
11    private LinkedBlockingQueue _pendingWrites = new LinkedBlockingQueue();
12    private Random _rand;
13
14    private Thread _readerThread;
15    private Thread _writerThread;
16
17    public ShellBolt(ShellComponent component) {
18        this(component.get_execution_command(), component.get_script());
19    }
```

```
20
21    public ShellBolt(String... command) {
22        _command = command;
23    }
24 }
```

- 第3行的_subprocess是利用ShellProcess加载的子进程。
- 第5行为目前消息的缓存。_inputs的键为一个随机值，用来唯一标识该消息，其值为消息本身。为了提高效率，Bolt会将接收到的消息通过发送线程不间断地发送给子进程。但由于从子进程收回的Ack或Fail等操作是异步的，因此Bolt需要根据随机的消息ID来获得原始消息，并对原始消息进行Ack或Fail操作。
- 第11行的_pendingWrites用来对那些要向子进程发送的消息进行缓存，Bolt的execute方法会将接收到的消息放置于该对列中，发送线程则读取该队列并将消息发送到子进程中。
- 第14~15行定义了读写线程。

27.2.2 读写线程

本节简要分析一下ShellBolt中的读写线程，相关代码如下：

```
1 public void prepare(Map stormConf, TopologyContext context,
2                     final OutputCollector collector) {
3     _rand = new Random();
4     _process = new ShellProcess(_command);
5     _collector = collector;
6
7     try {
8         //subprocesses must send their pid first thing
9         Number subpid = _process.launch(stormConf, context);
10        LOG.info("Launched subprocess with pid " + subpid);
11    } catch (IOException e) {
12        throw new RuntimeException("Error when launching multilang subprocess\n" + _process.
              getErrorsString(), e);
13    }
14
15    // reader
16    _readerThread = new Thread(new Runnable() {
17        public void run() {
18            while (_running) {
19                try {
20                    JSONObject action = _process.readMessage();
21                    if (action == null) {
22                        // ignore sync
23                    }
24
25                    String command = (String) action.get("command");
26                    if(command.equals("ack")) {
27                        handleAck(action);
28                    } else if (command.equals("fail")) {
29                        handleFail(action);
```

```
30                    } else if (command.equals("error")) {
31                        handleError(action);
32                    } else if (command.equals("log")) {
33                        String msg = (String) action.get("msg");
34                        LOG.info("Shell msg: " + msg);
35                    } else if (command.equals("emit")) {
36                        handleEmit(action);
37                    }
38                } catch (InterruptedException e) {
39                } catch (Throwable t) {
40                    die(t);
41                }
42            }
43        }
44    });
45
46    _readerThread.start();
47
48    _writerThread = new Thread(new Runnable() {
49        public void run() {
50            while (_running) {
51                try {
52                    Object write = _pendingWrites.poll(1, SECONDS);
53                    if (write != null) {
54                        _process.writeMessage(write);
55                    }
56                } catch (InterruptedException e) {
57                } catch (Throwable t) {
58                    die(t);
59                }
60            }
61        }
62    });
63
64    _writerThread.start();
65 }
```

- 第2~13行调用ShellProcess来启动子进程。
- 第16~44行定义读线程。根据从协议返回的action中获得的命令，ShellBolt将执行相关操作。
- 第48~62行定义写线程，即从_pendingWrites中获取一条消息并调用_process的writeMessage方法。
- 方法execute首先在第7行为输入的消息生成消息ID，然后在第8行将输入的消息放入_inputs中进行跟踪。
- 在第10~16行根据输入消息构建JSON对象，并将其放入_pendingWrites队列中，发送线程负责将消息发送出去。execute方法的代码如下：

```
1 public void execute(Tuple input) {
2     if (_exception != null) {
3         throw new RuntimeException(_exception);
4     }
```

```
5
6     //just need an id
7     String genId = Long.toString(_rand.nextLong());
8     _inputs.put(genId, input);
9     try {
10        JSONObject obj = new JSONObject();
11        obj.put("id", genId);
12        obj.put("comp", input.getSourceComponent());
13        obj.put("stream", input.getSourceStreamId());
14        obj.put("task", input.getSourceTask());
15        obj.put("tuple", input.getValues());
16        _pendingWrites.put(obj);
17    } catch(InterruptedException e) {
18        throw new RuntimeException("Error during multilang processing", e);
19    }
20 }
```

handleAck方法会对输入的消息进行Ack操作。它从action中获取消息ID，继而从_inputs中得到原始消息，然后就可调用ack方法。该方法的代码如下：

```
private void handleAck(Map action) {
    String id = (String) action.get("id");
    Tuple acked = _inputs.remove(id);
    if(acked==null) {
        throw new RuntimeException("Acked a non-existent or already acked/failed id: " + id);
    }
    _collector.ack(acked);
}
```

其他实现方法的原理都大体类似，这里不再赘述。

27.3　ShellSpout

ShellSpout的实现过程与ShellBolt的相近，本节简单讨论一下其协议部分，相关代码如下：

```
1 public void nextTuple() {
2     if (_next == null) {
3         _next = new JSONObject();
4         _next.put("command", "next");
5     }
6
7     querySubprocess(_next);
8 }
9
10 public void ack(Object msgId) {
11     if (_ack == null) {
12         _ack = new JSONObject();
13         _ack.put("command", "ack");
14     }
15
16     _ack.put("id", msgId);
```

```
17     querySubprocess(_ack);
18 }
19
20 private void querySubprocess(Object query) {
21     try {
22         _process.writeMessage(query);
23
24         while (true) {
25             JSONObject action = _process.readMessage();
26             String command = (String) action.get("command");
27             if (command.equals("sync")) {
28                 return;
29             } else if (command.equals("log")) {
30                 String msg = (String) action.get("msg");
31                 LOG.info("Shell msg: " + msg);
32             } else if (command.equals("emit")) {
33                 String stream = (String) action.get("stream");
34                 if (stream == null) stream = Utils.DEFAULT_STREAM_ID;
35                 Long task = (Long) action.get("task");
36                 List<Object> tuple = (List) action.get("tuple");
37                 Object messageId = (Object) action.get("id");
38                 if (task == null) {
39                     List<Integer> outtasks = _collector.emit(stream, tuple, messageId);
40                     Object need_task_ids = action.get("need_task_ids");
41                     if (need_task_ids == null || ((Boolean) need_task_ids).booleanValue()) {
42                         _process.writeMessage(outtasks);
43                     }
44                 } else {
45                     _collector.emitDirect((int)task.longValue(), stream, tuple, messageId);
46                 }
47             }
48         }
49     } catch (IOException e) {
50         throw new RuntimeException(e);
51     }
52 }
```

- 第20~52行定义的querySubprocess是ShellSpout的核心。它在第22行向子进程写入命令，然后不断接收消息，直到收到sync命令。收到emit命令时，querySubprocess将从action中获得消息tuple、目标taskId（Direct Emitting）和消息跟踪序号messageId。可见消息的跟踪序号是由子进程产生的，子进程需要维护从消息序号到消息的对应关系，ShellSpout则并不维护该映射关系。
- 第1~8行定义nextTuple函数，它会发送next命令到子进程，并调用querySubProcess函数。
- 第10~18行定义ack函数，即将ack命令和消息跟踪序号发送给子进程进行处理。

第 28 章 Storm中的配置项

Storm的配置数存储在Config. java文件的一个哈希表中，这里每个参数都有对应的变量名，变量名到defaults.yaml中定义的配置项的转换规则为将字母转成小写，并把“_”替换为“.”。例如：

```
public static String STORM_ZOOKEEPER_SERVERS = "storm.zookeeper.servers";
```

若要在Clojure代码中访问这些变量，则映射规则为将“_”替换成为“-”。例如：

```
STORM-ZOOKEEPER-SERVERS
```

表28-1为Storm中的配置项参数，它们的默认值可通过defaults.yaml来调整。

表28-1　Storm的配置项

参　　数	默　认　值	含　　义
dev.zookeeper.path	/tmp/dev-storm-zookeeper	N/A
drpc.invocations.port	3773	被DRPC Topology使用，用来获取DRPC查询请求并发送结果的端口
drpc.port	3772	DRPC服务器上客户端用于发送请求并获取结果的端口
drpc.queue.size	128	DRPC Thrift服务器中请求队列的大小
drpc.request.timeout.secs	600	DRPC请求的超时时间
drpc.worker.threads	64	DRPC服务器的工作线程数目
java.library.path	sbin	JVM的Lib查找路径
nimbus.childopts	-Xmx1024m	Nimbus的JVM配置
nimbus.cleanup.inbox.freq.secs	600	Nimbus收件箱清理的时间间隔
nimbus.file.copy.expiration.secs	600	下载上传的超时时间
nimbus.inbox.jar.expiration.secs	3600	Jar文件在收件箱的存活时间
nimbus.monitor.freq.secs	10	心跳检查和任务分配的时间间隔
nimbus.reassign	true	表明检测到任务失败后是否重新分配任务
nimbus.supervisor.timeout.secs	60	Supervisor的心跳超时间隔
nimbus.task.launch.secs	120	第一次启动Task时的超时设置

（续）

参　　数	默 认 值	含　　义
nimbus.task.timeout.secs	30	Task启动的超时设置
nimbus.thrift.port	6627	Nimbus的服务器端口号
nimbus.topology.validator	backtype.storm.nimbus.DefaultTopologyValidator	配置Topology合理性检查插件
storm.cluster.mode	distributed	Storm的运行模式，或者为（lacal）模拟，或者为分布式 模拟模式需利用LocalCluster类来运行
storm.id	N/A	Topology的名字，即用户提供的名字+一个唯一标识ID
storm.local.dir	D:/Data/Storm/stormData	Storm用来存储本地数据的路径
storm.local.mode.zmq	false	表明是否在LocalCluster模式下使用ZMQ
storm.zookeeper.connection.timeout	15000	ZooKeeper的连接超时时间
storm.zookeeper.port	2181	ZooKeeper服务器的端口号
storm.zookeeper.retry.interval	1000	ZooKeeper操作的重试间隔
storm.zookeeper.retry.intervalceiling	30000	ZooKeeper的最大重试间隔（未使用）
storm.zookeeper.retry.times	5	ZooKeeper操作的重试次数
storm.zookeeper.root	/storm	Storm元数据的ZooKeeper根目录
storm.zookeeper.servers	["xx.xx.xx.000"	ZooKeeper的服务器IP地址
storm.zookeeper.session.timeout	20000	ZooKeeper客户端的超时时间
supervisor.childopts	-Xmx256m	JVM参数，Worker的内存限制
supervisor.enable	true	表明是否使用该Supervisor，用于测试
supervisor.heartbeat.frequency.secs	5	Supervisor的心跳时间间隔
supervisor.monitor.frequency.secs	3	Supervisor检查Worker心跳的时间间隔
supervisor.slots.ports	[6700~6703]	Supervisor中Worker所对应的端口号，每一个端口号对应一个Worker，故Supervisor最多可以启动端口号数目个Worker
supervisor.worker.start.timeout.secs	120	Worker第一次启动时的超时时间间隔 第一次启动JVM时会存在额外负载，故长于supervisor.worker.timeout.secs的设置
supervisor.worker.timeout.secs	30	Worker的超时时间间隔，若Supervisor在该时间内未收到Worker的心跳，则重启Worker
task.heartbeat.frequency.secs	3	Task发送心跳到ZooKeeper的时间间隔
task.refresh.poll.secs	10	ZMQ连接更新时间以及Topology是否活跃的查询时间
topology.acker.executors	1	系统中Acker Bolt的数目。若为0，Spout将直接对发送的消息进行Ack

（续）

参　数	默 认 值	含　义
topology.acker.tasks	N/A	N/A
topology.builtin.metrics.bucket.size.secs	60	内置统计信息的统计时间间隔
topology.debug	false	表明是否对Topology中发送的消息进行打印
topology.disruptor.wait.strategy	com.lmax.disruptor.BlockingWaitStrategy	Disruptor Queue的等待策略，默认为等待10毫秒，若没有消息则返回
topology.enable.message.timeouts	true	表明是否允许消息超时。若为假，则Spout节点不会产生System Tick消息，Spout的Pending的消息的超时方法也便不会被调用，因此消息不会超时
topology.error.throttle.interval.secs	10	Storm中利用ZooKeeper来报告错误。Storm规定在特定的时间区间内，只发送不超过一定数目的错误，该参数用来设定这个数目。它与参数topology.max.error.report.per.interval一起使用
topology.executor.receive.buffer.size	1024	Executor接收消息Disruptor Queue的大小
topology.executor.send.buffer.size	1024	Executor发送消息Disruptor Queue的大小
topology.fall.back.on.java.serialization	true	表明Kryo序列化失败时，是否使用Java默认的序列化方法
topology.kryo.decorators	[]	用于定制Kryo序列化实例，其中的类需要实现IKryoDecorator接口
topology.kryo.factory	backtype.storm.serialization.DefaultKryoFactory	用于产生Kryo序列化器的工厂方法
topology.kryo.register	N/A	用户注册的序列化类
topology.max.error.report.per.interval	5	Storm中利用ZooKeeper来报告错误。Storm规定在特定的时间区间内，发送不超过一定数目的错误，该参数用来设定时间区间。与参数topology.error.throttle.interval.secs 一 起使用
topology.max.spout.pending	1	Spout中最大可以处于Pending状态的消息数目 消息的Pending是指消息已经发送出去，但未收到Ack或者Fail 处于Pending的消息超出该设置后，Spout将不再发送消息。 该配置项是Task级别的。例如，一个Spout的并行度为10，该参数为10，则所有Spout最多可以有100条处于Pending状态的消息
topology.max.task.parallelism		组件的最大并行度
topology.message.timeout.secs	120	消息的超时时间设置。若Spout发送出去的消息未在该时间内被Ack，Storm会将消息标记为失败，某些Spout可以进行重传或者将消息忽略

（续）

参　数	默 认 值	含　义
topology.name	N/A	Topology的名字，在提交Topology时被自动设置
topology.optimize	true	N/A
topology.receiver.buffer.size	8	从ZMQ接收消息的批的大小，当收到该数目的消息后，将分发消息到Executor的接收队列
topology.skip.missing.kryo.registrations	false	当Kryo需要不能识别的类或者序列化器时，是否加载Kryo注册的序列化器，否则在遇到该情况时将抛出异常
topology.sleep.spout.wait.strategy.time.ms	1	SleepEmptyEmitStrategy 的睡眠时间。当nextTuple未发送出去消息时，则令线程睡眠该时间
topology.spout.wait.strategy	backtype.storm.spout.SleepSpoutWaitStrategy	和上面参数结合使用，为当Spout的nextTuple未发送消息时的处理策略
topology.state.synchronization.timeout.secs	60	Topology元数据的同步时间
topology.stats.sample.rate	0.05	运行统计的采样频率
topology.tasks		该参数与组件相关，表示组件的并行度
topology.tick.tuple.freq.secs		Tick消息的发送时间间隔
topology.transfer.buffer.size	1024	Executor发送消息队列的大小
topology.trident.batch.emit.interval.millis	500	Trident的Batch产生时间间隔。在过多Batch被发送时用来控制流量
		被类WindowedTimeThrottler使用
topology.worker.childopts		Worker启动的子进程的JVM参数（未使用）
topology.worker.shared.thread.pool.size	4	Worker中线程池的大小
topology.workers	4	与Topology相关，表明希望使用多少个Worker来执行Topology
transactional.zookeeper.port		用于存储事务的ZooKeeper端口号
transactional.zookeeper.root	/transactional	用于存储事务的ZooKeeper数据的根目录
transactional.zookeeper.servers		用于存储事务的ZooKeeper服务器
ui.childopts	-Xmx768m	Storm UI的JVM配置
ui.port	83	Storm UI的端口号
worker.childopts	-Xmx16g	Worker的JVM配置
worker.heartbeat.frequency.secs	1	Worker心跳的时间间隔
zmq.hwm	0	ZMQ的发送队列的高水平线，0表示不设置
zmq.linger.millis	5000	当队列中有未发送的消息时，ZMQ的关闭等待时间
zmq.threads	1	ZMQ的线程数



“本书从源代码角度深入浅出地分析了Storm的设计及实现，一方面可以使读者更好地了解并用好Storm技术，另一方面可以让读者学习如何设计大规模分布式系统，相信读者一定会受益匪浅。”

—— 于伟，微软资深开发总监

“书中对Storm 的理解精辟透彻，对Storm的运用和各处细节也都阐述入微。尤其是对Storm的入门初学者来说，是一本不可多得的好书。”

—— 章英基，前微软资深开发总监，现阿里巴巴资深总监

“本书由微软公司互联网工程院经验丰富的一线程序员操刀编写，包含很多实战经验和使用心得，很好地结合了代码分析和应用实例。本书对于进行流式数据处理的研究、Storm的深入理解以及实际应用都有很好的参考价值。”

—— 王明雨，微软资深开发工程师

“在工作期间，这本书对我帮助很大，即便对于像我这样在分布式领域工作12年的老手来讲，这本书仍然让我受益良多。无论你是大数据领域、分布式系统的从业人员，还是开源系统的爱好者、开发者或互联网从业人员，我认为这本书都值得仔细研读。”

—— 贺军，微软资深项目经理

“本书从源代码的角度深入解读了Storm技术。两位作者为微软公司互联网工程院的一线程序员，拥有丰富的实战经验。本书不仅可以让你全面了解Storm工作原理，深入洞悉Storm底层架构，还有助于你学习如何设计大规模分布式系统。”

—— 熊平，51CTO传媒总裁

51CTO.com
技术成就梦想

图灵社区：iTuring.cn
热线：(010) 51095186 转 600

分类建议 计算机/大数据

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-37126-3



ISBN 978-7-115-37126-3

定价：79.00元